Reverse Engineering Legacy User Interfaces Using Interaction Traces

Mohammad El-Ramly

Ph.D Thesis

University of Alberta

2003

Chapter One

Introduction

"*By the time you will finish your thesis, the systems, tools and/or prototypes that you have developed will be legacy systems and you will need to reverse engineer them in order to understand, migrate and/or reengineer them.*"

Anonymous

1.1 Background

Over years of development and investment, business software systems, such as bank finance systems, customer relationship management (CRM) systems and airline reservation systems, grew in size and value. They contain the specifications for diverse business policies and corporate decisions and constitute some of the most important industrial assets for many companies [LBS94]. Corporations have invested substantially in developing these mainframe-based systems. They have almost spent as much to develop integrated reliable database systems. In the recent past, they invested even more money in making their systems Y2K and Euro compliant. [Sne00]

Many such systems were developed using the technology of the 1970s to mid-1980s. They have been modified many times by different programmers. As a result, they have

become very complex and difficult to understand, maintain, renovate and/or reengineer [LBS94]. Such systems are referred to in the literature as "legacy systems". The Free Online Dictionary of Computing [FOLD96] defines a legacy system as:

Definition 1.1

> *"A computer system or application program which continues to be used because of the prohibitive cost of replacing or redesigning it and despite its poor competitiveness and compatibility with modern equivalents. The implication is that the system is large, monolithic and difficult to modify."*

This definition includes hardware and software systems and is neither restricted to mainframe-based systems nor to a specific programming language or platform. Gold [Gold98] expands the definition of a legacy system to include not just the hardware and software but also the environment, the people, the procedures, etc., surrounding the system. He states:

Definition 1.2

> *"A legacy system is a socio-technical system containing legacy software"*

This is because when a critical software system ages, not only understanding and modifying the system becomes hard, but also changing and modifying the surrounding environment, especially the people, becomes hard too. Bergey *et al.* [BSTWW99] consider resistance to change and the growth of a culture dependent on maintaining the status quo plus inadequate training programs as the third reason for failure of reengineering projects. This thesis adopts the first definition, but it is only concerned with software systems not hardware.

In return for the effort and investment spent, mainframe-based legacy systems have demonstrated robustness, reliability and scalability in providing business-critical processing needs. This is especially true where the application concerned involves huge numbers of transactions and many simultaneous users, as is the case with banking or airline reservation systems. Most important, however, is that many of the business processes of companies are encapsulated in the logic of legacy applications; they are, in effect, the repositories of hard-won corporate experience and knowledge, that may not be

available in other formats [Att00]. Considering this, legacy systems will remain the Information Technology (IT) backbone for many corporations, for many years to come.

However, due to their age, many legacy systems suffer from some or all of the following aging disadvantages [Par94]:

1. Performance and functionality degradation.
2. Lack of coding standards, proper documentation and version control.
3. Incremental and patch updates to the code and design that often violate the original software design concepts. These "ignorant surgeries", as Parnas [Par94] calls them, result in degradation of the maintainability and comprehensibility of the software.
4. Pollution [Vis01], which is the accumulation of duplicate and dead code (that is either never compiled or never executed), useless components (e.g., reports that no user needs anymore), and dead data.
5. A general lack of understanding of the internal workings of the system and how its functions relate to its modules and data.
6. Significant resistance to modification and evolution, not only due to technical difficulties but also due to socio-political factors.
7. Character-based user interfaces (CUIs), which are not competitive with the superior alternatives offered by today's technology.
8. Great difficulty in integrating with other systems and the World-Wide-Web (WWW).

Because of these symptoms, many organizations are migrating, renovating or reengineering their legacy systems to achieve one or more of the following objectives:

1. Migrating the whole application to a newer, faster, and non-proprietary platform.
2. Enhancing system comprehensibility and maintainability, i.e., putting it under control.
3. Adding new substantial or minor functionality.
4. Integrating the legacy application with other applications, on legacy or new platforms.
5. Migrating the application user interface (UI) to a new platform.
6. Enabling access to the system through the WWW.

Depending on the goal of the reengineering effort and the current status of the legacy system, reengineering activities can vary widely. They range from rebuilding the system and using the legacy system as an input for the analysis and design phase, to wrapping

the legacy system to fit it in a new computing environment, e.g., graphical user interfaces (GUIs), CORBA, client/server architectures, or the WWW.

1.2 Motivation

When the objective of a legacy system reengineering effort is migrating its UI to a new platform, enabling access to the system through the Web (Web-enabling) or lightweight front-end integration with other systems, then the reengineering effort can focus on enhancing the UI of the legacy system or developing a new UI. This is especially applicable, when the prime aging symptoms of the legacy systems are the last two of the eight mentioned above, or in other words, the legacy system is under control and exhibits satisfactory performance but its main weakness is its poor UI and its inability to be integrated with other systems. This occurs due to the legacy UI falling short in three areas: user access, usability and navigation at different levels [BB01]:

1. **User Access.** Most legacy systems are proprietary monolithic systems that were not designed with integration with the WWW or other technologies in mind. Usually they do not have clear separation between their presentation, logic and data layers, which makes opening a legacy system for access via a new platform or for front-end integration with other systems a hard task. The presentation layer refers to the source code that controls the UI. The logic layer refers to the application code that provides the main functionality. The data layer refers to the container of the application data and the code used to access it.
2. **Usability.** The old-looking "dumb" terminals, e.g., IBM 3270 and VT series, were quite adequate for their time in spite of being quite limited in their display capabilities. Legacy character-based UIs are non-intuitive and hard to learn. Their UIs dissatisfy today's users, who are used to graphical user interfaces and Web interfaces. Additionally, the learning curve of new users is slow and the training costs are high.
3. **Navigation.** Due to their limited presentation capabilities, legacy character-based UIs offer tedious navigation patterns to accomplish user tasks. For example, flipping a multi-page report may require using function keys or issuing some commands to move forward and backward between the many screens containing the report. Instead, in a GUI environment, a scroll bar enables instant access to any page of the report with a mouse click.

So, in many cases it is unnecessary, hard, expensive, risky and/or impossible to change the code of a legacy system and design, yet, it is desired to reengineer its UI. The goal of this reengineering is to open the system to the Web, Wireless Access Protocol (WAP) or other platforms, to integrate its front-end with those of other systems and/or to slightly extend its functionality. For these cases, there is a need for lightweight non-invasive UI reengineering methods. These methods need to be lightweight in the sense that they are cost-effective, semi-automated and relatively easy to deploy because they require moderate skills and low technology. And they are non-invasive in the sense that they almost do not need alteration of the legacy system. Industry has offered some primitive labor-intensive solutions to this problem.

The CelLEST project for UI reengineering [SES02, SEIS03] is a collaborative project between the Software Engineering Research Lab. at University of Alberta and an industrial partner; Celcorp [Cel]. The goal of CelLEST was to develop an intelligent semi-automated lightweight non-invasive method and prototype tools for legacy system CUI reengineering, Web-enabling and front-end integration. The method developed in this project takes as input recorded traces of interaction between the legacy system and its users through the legacy CUI (interaction traces) and does not require modifications to the legacy code. We call this approach to UI reengineering "interaction reengineering" as opposed to "code reengineering". The CelLEST interaction-reengineering method consists of two phases: a reverse engineering phase and a forward engineering phase. The focus of this thesis is the reverse engineering phase. It describes the novel legacy CUI reverse engineering method developed, during and after the CelLEST project. Since this method takes interaction traces as its only input, in effect, it adopts an "interaction reverse engineering" approach as opposed to "code reverse engineering".

1.3 The CelLEST Project

To achieve its goal of developing an intelligent semi-automated lightweight non-invasive method for CUI reengineering, CelLEST employs a mixture of document analysis, clustering, example-based modeling of user actions, visualization, data mining, task model inference, XML wrapping and automated GUI layout. The outcome is a novel CUI reengineering method that utilizes interaction traces and does not change the legacy system code or structure. The CelLEST method, resulting from the project, supersedes

the current manual, labor intensive practices of legacy character-based UI modeling and reengineering, which need intensive human input, intuition and experience.

When invasive solutions are unnecessary, undesirable, too hard, impossible, risky and/or cost-ineffective and code and architecture modification is not mandatory for front-end reengineering of a legacy system, the CelLEST method is a very suitable solution. Since it is important to understand CelLEST method to understand the context of this thesis, the method is introduced briefly in this section and is described in more detail in chapter 3.

1.3.1 Two CelLEST Reengineering Scenarios

To fully understand and appreciate the motivation behind the CelLEST UI reengineering project, this subsection describes two typical legacy system CUI reengineering scenarios that are most suitable for the application of CelLEST method. These scenarios faithfully represent real cases encountered by practitioners

1.3.1.1 Migrating a Students Information System (SIS) to the WWW

In this scenario, an educational institute developed its student information system (SIS) in the mid-1980s. SIS was written in COBOL and NATURAL (a 4GL) running on the institute's IBM Mainframe platform. SIS included modules like course catalog, schedule of classes, admissions, student biographic data, registration, and academic history. SIS interacts with the Account Receivable Information System, which handles student payments. It also interfaces with a phone registration system. The primary users of SIS are the employees of the Registrar’s Office and the students who can only access the registration module using the phone registration system. Additionally, SIS provides system-wide managerial and student information on student enrollment and activity to the administrators to assist them in planning and decision-making on an institution-wide basis. The status quo of the system was quite satisfactory for the management. Its performance, reliability and scalability to high workload at peak times of the year were very good. Additionally, the employees were well trained on the existing character-based UI. System maintainers were familiar with its design and code.

In the late 1990s, the institute wanted to use the Internet to allow students access to student services directly rather than having to go through administrators. Therefore, students could have self-service access to information on enrollment, timetables, grades,

and various financial accounts via the WWW. This would free the institute's administrative staff from repetitive and routine tasks and significantly reduce administrative costs.

A UI reengineering solution was needed to open the system to Internet access and to provide an easier alternative to the current character-based UI, which is not easy to use for the general student population. Only tasks relevant to students would be opened to the students. Management ruled out any invasive solution that would involve modifying the existing system due to the cost and risks involved. They also ruled out any solution that would involve duplicating the existing system application logic or data, to avoid possible inconsistencies and extra maintenance overhead. For such a scenario, the CelLEST method is an excellent solution. It can leverage the existing CUI to a Web UI without modifying the existing system. Semi-automatically, CelLEST can generate Web-based wrappers of the desired tasks. CelLEST is an incremental solution that can be implemented and tested gradually by wrapping a small selected number of student tasks and trying them, then wrapping and Web-enabling more functionality, etc.

1.3.1.2 Integrating the Front-ends of Two Insurance Systems

In this scenario, an insurance company acquired another insurance company. Both companies had similar information systems for claims. The performance of each system was satisfactory when they were under different ownership. After the merger, the new owner did not want its employees to use two independent systems, which incurs additional training costs and productivity reduction due to effort duplication and the time consumed in switching between systems. Merging both systems via reengineering or transferring the data of one system to the other was infeasible due to the technical difficulties, prohibitive cost and risk involved. A suitable solution for integration in this case would be CelLEST method. It can provide lightweight front-end integration under a unified task-centered GUI that abstracts both systems' UIs. Additionally, it was required to offer easy access of both systems via an extranet to lawyers who handle cases involving insurance claims. A limited tailored version of the new GUI can be offered to lawyers, which offers specific lawyer-oriented tasks. Since these lawyer-oriented tasks were not required at the time of system development, they were not directly achievable through the existing UIs but the bits and pieces needed for each task were scattered in the

character-based UIs. The tailored GUI accomplishes these tasks by taking the necessary inputs from the lawyer in a format that is most natural for him/her and driving the necessary navigation in the two legacy UIs to reach the right screens to gather the needed outputs. Then, it reformats these outputs in the format most natural for lawyers and presents them via the target GUI platform.

1.3.2 The CelLEST Process

Building on the two example scenarios given above, let us now discuss the methodological assumptions of the CelLEST method:

1. The current performance of the legacy system subject to reengineering is satisfactory and it will continue to be used on its current platform.
2. It is required to open the system for access through a new platform, e.g., the WWW or WAP devices or window-based systems, or to integrate the front-end of the legacy system with other applications on the same or other platforms.
3. It is required to take advantage of the presentation and navigation capabilities of the target platform. So, each user task will be encapsulated, with all the input, output and navigation steps required to accomplish it, in a suitable task-oriented representation on the target platform, e.g., a number of Web-forms. In other words, the new UI should be "task-centered".
4. No major functionality change is required. However, minor functionality may be added if it is achievable based on the data presented on the original interface.
5. It is undesirable or impossible to change the legacy system code and architecture. The main input to the UI reengineering process will be recorded traces of interaction between the legacy system UI and its users, while they are doing their regular tasks.

The CelLEST process is a two-phase process. The first is a reverse engineering phase, which is the focus of this thesis [SEKSM99, EISSM01, SEIS03, SES02, ESS02b ESS02c]. The algorithms and methods developed to support the reverse engineering phase and their evaluation are presented in details in this thesis. They are implemented in a prototype tool called the ***Le**gacy **N**avigation **D**omain **I**dentifier* (LeNDI). The second is a forward engineering phase, which was conducted by other members of CelLEST research team [KSM99, Kap01, SK02]. The algorithms and methods developed to

support the forward engineering phase are implemented in a prototype tool called *Mathaino*.

1.3.2.1 CelLEST Character-base User Interface Reverse Engineering

First, LeNDI is used to record the dialogs that take place between the legacy system and its users while they are doing their tasks in the form of interaction traces. An interaction trace is a sequence of legacy screen snapshots interleaved with the user actions done to cause the transitions between these snapshots. A user action is a sequence of keystrokes. Throughout this thesis, the term "screen" is used to refer to a CUI behavioral state manifested by a matrix of characters displayed to the user on her/his terminal, which allows her/him to do one of a limited set of actions. The term "snapshot" is used to refer to an instance of a screen. One can think of screens and snapshots as classes and objects in object-oriented terms.

Second, for every snapshot recorded in the interaction traces, LeNDI extracts a set of features and employs interactive clustering, classifier induction and user action modeling methods to build a behavioral model of the legacy CUI, called the state-transition graph. The nodes of this model correspond to the CUI behavioral states, i.e., screens and the edges correspond to the user actions causing transitions among the nodes. LeNDI utilizes two clustering methods to group similar snapshots together as one legacy CUI screen modeled by one node on the graph. Then, LeNDI infers a predicate that identifies the snapshots of the screen. LeNDI uses the user actions recorded in the interaction traces to model the behavior of the legacy screens as the arcs of the directed graph. The state-transition graph is a main input to the forward engineering phase of CelLEST. It is used to classify each individual snapshot forwarded by the legacy system to the user while s/he is interacting with it online.

Third, LeNDI uses data-mining methods to discover patterns of frequent segments of interaction between the legacy system and its users which correspond to popular usage scenarios of the system, or in other words the most used services of the system. We call such patterns "interaction patterns". The instances of an interaction pattern may have some noise due to spurious navigation of the legacy CUI. LeNDI interaction pattern mining algorithms can tolerate a preset level of noise and still discover patterns with this level of noise. An interactive review and revision process is associated with the behavior

modeling and pattern mining processes. This is to give the user control over these processes on one hand, and to get his feedback to verify the correctness of the models and the usefulness of the patterns produced, on the other hand.

1.3.2.2 CelLEST Character-base User Interface Forward Engineering and Visualization

In the forward engineering phase of CelLEST, Mathaino is used to augment each interaction pattern discovered with the semantic information needed to build a model of the task. Then, Mathaino is used to construct a declarative user-interface specification for the modeled task. This specification is also executable by a suite of special-purpose platform-specific components. Thus, the new user-interface becomes a front-end for the original legacy user-interface, available in multiple new platforms, e.g., XHTML-enabled browsers or WAP devices. The new interface executes a task in the underlying legacy application using the state-transition model of the application's CUI, a model of the task, and an API to the data-transfer protocol used by the legacy system.

In addition to LeNDI and Mathaino, the CelLEST environment includes the QandA (Questions AND Answers) tool [Vij02], which supports the visualization, verification and possibly revision of all the intermediate products of the CelLEST method by the analyst.

1.3.3 Advantages of the CelLEST Process

The CelLEST method is a significant contribution to the field of CUI reengineering. This is because it has a number of advantages:

1. It is a code-independent non-invasive CUI reengineering method. It utilizes easy to collect input, i.e., interaction traces. So, it is very suitable when code modification is undesirable, expensive, risky or impossible. The limitation of this method is that it can support only minor functionality extensions.
2. CelLEST is lightweight in terms of the skills it assumes. It needs moderate analysis skills and an understanding of the system under analysis as opposed to the high software development skills and expert understanding of the legacy system that current practices demand. It is lightweight in terms of the cost and time. Therefore, it can potentially bring substantial time and cost reduction and quality improvement to current state-of-the-art industry practices.

3. CelLEST is an incremental approach. It can be applied gradually to the exiting legacy CUI. Thus, a phased reengineering can take place with some services of legacy system CUI reengineered in every phase, according to the time and budget available.
4. CelLEST follows a task-centered approach to reengineering the way the legacy system users interact with it. It encapsulates interesting behavioral segments within new UI front-ends on different platforms. It does not replicate the legacy system-user interaction with different widgets in new platforms.
5. CelLEST supports simultaneous migration to multiple platforms. It does CUI reverse and forward engineering once and generates abstract specifications of the reengineered task-centered UI. A new UI can be generated multiple times on different platforms using the abstract UI specifications.
6. CelLEST emerged from collaboration with industry on one hand and on the other hand, it was developed and evaluated in an academic setup, with very encouraging results. This mixture of research and industry gives CelLEST the potential to impact current CUI reengineering practices on solid scientific bases.

1.4 Thesis Statement

This thesis makes a case for automated CUI interaction reverse engineering. It takes the position that recorded traces of interaction between the users of a legacy system with its character-based user interface can be sufficient input for lightweight non-invasive reengineering of the user interface. It demonstrates that reverse engineering these interaction traces can provide the CUI behavioral model required for the reengineering process and shows how the elements of this model can be inferred from these traces. Additionally, it demonstrates that patterns of user activities with the legacy CUI can be discovered from the interaction traces, and used as a basis for identifying and modeling the system services that are candidate for reengineering. Finally, this thesis demonstrates, via case studies, the practicality, efficiency and usefulness of the automated CUI interaction reverse engineering process, and hence, proves its potential impact on advancing the current manual practices for lightweight CUI reengineering and Web-enabling.

1.5 Thesis Contributions

This thesis establishes a novel method for CUI reverse engineering that adopts interaction reverse engineering as the means to build a behavioral model for the legacy CUI and to capture the interesting user interaction patterns with the system. The behavioral model and the interaction patterns are used for CUI reengineering. The specific contributions of this thesis are:

1. **Engineering a suite of features for characterizing CUI screen snapshots [SEKSM99, SEIS03]**. These features are extracted from analysis of the snapshots with a set of tailored heuristics and document analysis methods. This analysis extracts features from any special information discovered at the periphery of the snapshot, from the hidden snapshot information coming with the outbound data streams received from the host, and from the snapshot layout and content distribution. Some of these features are specific to IBM 3270 data streams, but most of them are applicable to other block-mode data transfer protocols, which push one screen at a time to the user (as opposed to scroll-mode data transfer protocols, which interact with the user line by line).
2. **An intelligent semi-automated method for modeling the behavior of legacy CUIs [EISSM01, SEIS03]**. This method is a significant advance to the research and practice of CUI reengineering. It consists of the following steps:
   - Recording traces of interaction between the legacy system CUI and its users.
   - Extracting a feature vector for every recorded snapshot.
   - Clustering similar snapshots together, based on their feature vectors similarity.
   - Inferring predicates for each cluster (i.e., each distinct CUI state) via classifier induction.
   - Example-based learning of the user actions that cause transitions from one state to another.
   - Mining the interaction traces for patterns of user interaction with the legacy CUI.
3. **Two novel sequential pattern mining algorithms [ESS02b, ESS02c].** IPM is a breadth-first algorithm and IPM2 is a depth-first algorithm. Although, both algorithms are designed specifically to mine interaction traces for interaction patterns, they can mine sequential data in general for sequential patterns with noise.

4. **A prototype tool for interaction-based CUI reverse engineering, called LeNDI [SES02, SEIS03]**. LeNDI implements all the methods and algorithms described in this thesis. LeNDI was used to evaluate the CUI reverse-engineering method with case studies from real legacy systems, with very promising results.

1.6 Thesis Organization

The rest of this thesis is organized as follows:

- Chapter 2, *Related Work*, introduces the related research areas. It covers four areas of research that represent the broader areas of this thesis and the state-of-the-art in research and industry of UI reengineering and Web-enabling.
- Chapter 3, *CelLEST User Interface Reengineering*, is a detailed description of the CelLEST method for CUI reengineering at large. It helps the reader understand the context, in which, the research described in this dissertation was conducted.
- Chapter 4, *Feature Extraction For Legacy Screen Snapshots*, describes the feature suite engineered in LeNDI for characterizing the snapshots of CUI screens. It details the process of extracting a feature vector for every recorded snapshot and the algorithms used in it.
- Chapter 5, *Legacy User Interface Behavior Modeling*, describes the process and algorithms used to identify the nodes and arcs of the state-transition model of a legacy CUI, and consequently build it. Additionally, it discusses the experiments and case studies conducted to evaluate this process and the associated algorithms.
- Chapter 6, *Mining Interaction Traces for Patterns of Frequent User Tasks*, describes the process and the two novel algorithms (IPM and IPM2) used to mine the interaction traces for interaction patterns. It provides evaluation for this process and comparison of the performance of IPM and IPM2.
- Chapter 7, *Summary, Conclusions and Future Work*, presents a summary of this dissertation, draws some conclusion and points to future work directions.

Chapter Two
Related Work

This chapter describes four areas of research related to the research described in this thesis. The diversity of these areas reflects the diversity of the methods and algorithms used in this research. The first area is *UI reengineering and reverse engineering*. The second area is driven by practice and industry more than by academia, which is *Web-enabling legacy systems*. The third area is *software requirements recovery* from available legacy resources. The forth area is *data mining of sequential data* to discover sequential patterns in long sequences of data. Each of the four sections of this chapter is dedicated to one of these areas. Each section starts by a definition of the research area, followed by the motivation behind studying this area and by a description of representative, state-of-the-art work. Finally, it concludes with comments on how this work is similar to or different from the work in this thesis.

Section 2.1 is a review of the work in the area of UI reengineering and reverse engineering. The CelLEST project is about UI reengineering, but is different than other work in this area in that it adopts a lightweight "don't touch code" approach for UI reengineering. LeNDI adopts a novel UI reverse reengineering method using interaction traces, designed to suit the interaction reengineering approach of CelLEST

Section 2.2 describes the state-of-the-art industrial practices of Web-enabling legacy systems, by uncovering and accessing the logic, data or presentation layers of the legacy system. In particular it focuses on the current manual practices of reverse engineering legacy CUIs using presentation layer access [Ake00, Att00, Cri01] and describes how LeNDI supersedes and advances these manual practices by providing a coherent automated process that is more efficient and less susceptible to error. The area of Web-enabling legacy systems is described in more detail since it is the most related area to this work. In addition, since a lot of Web-enabling work is coming from industry not academia, detailed discussion of the evolution of this area, that relates different practices to each other, was needed.

Section 2.3 describes the related work in the area of software-requirements recovery. It describes the different methods used or proposed in literature for recovering the functional requirements or specifications of software systems, using different inputs. LeNDI employs a novel automated method for recapturing models of the current uses of a legacy system as its *de facto* functional requirements. LeNDI's method is easy, practical and does not assume the availability of system code or documentation. While the recovered models are needed for the CelLEST forward engineering phase, they can be translated to use case models and utilized in different contexts.

Section 2.4 describes the related problems and significant work in the area of sequential pattern mining, inspired by data mining and bioinformatics applications. It explains the problem of interaction pattern mining in LeNDI, which is mining the traces of interaction with a legacy system for interaction patterns. It describes how this problem is different from other problems and hence, why LeNDI needs a new algorithm to solve it. This led to developing two novel algorithms for interaction pattern mining, Interaction Pattern Miner (IPM) and Interaction Pattern Miner 2 (IPM2), which are described in details in chapter 6.

2.1 User Interface Reengineering and Reverse Engineering

UI Reengineering is the process of recreating an existing UI, either on the same or a different platform. In some cases, UI reengineering is done as part of legacy system migration to a new platform. In other cases, UI reengineering is done for itself to reface an existing legacy application with or without minimal changes to the system.

UI Reengineering is a two-step process. The first is a reverse engineering phase, during which an abstract representation of the legacy UI is created. The second is a forward engineering phase, during which, a new or modified implementation of the legacy UI is developed, usually on a new platform. Reverse engineering an existing UI can be desired in order to better understand an existing system. In such cases, the objective is to produce an abstract representation of the legacy UI to enhance the understandability and maintainability of the legacy system, especially its presentation layer. LeNDI is a CUI reverse engineering tool that uses, as input, recorded traces of interaction with the legacy system. Mathaino is a CUI forward engineering tool that uses the models developed by LeNDI to develop new GUIs, Web-interfaces or WAP-

interfaces for legacy systems. In developing LeNDI, the significant work on UI reengineering and reverse engineering was reviewed as follows shortly. None of the work described adopts the novel approach of interaction reverse engineering employed in LeNDI. LeNDI avoids code analysis and takes advantage of the relatively easy to collect interaction traces to deduce the legacy UI behavior models and interaction patterns needed in CelLEST.

Reengineering UIs of legacy systems can have different variants:

- UI full or partial redesign and re-implementation on the same platform [PRSV97].
- GUI to GUI migration due to platform change [MRS94].
- Character-based UI (CUI) to GUI migration [AFMT95].
- GUI grafting onto a batch or command-line system [PA97, TS99, WJD01, SCT02].
- Wrapping a legacy UI with a Web or WAP wrapper [Hor98, TLRH98, BFM02].

An example of the first category is the work of Plaisant *et al.* [PRSV97]. They employed different techniques to evaluate existing UIs for six different systems. These techniques include documentation study, observing users, expert review of the current UI, questionnaires and discussions with users and managers. For each system, they identified the main problems in its UI, if any, and the opportunities of improvement in user documentation, system access, data display, data entry procedures, consistency and error/system messages. These findings were used to partially re-implement the examined UIs on the same platform to improve user performance and satisfaction.

An example of the second category is the work of Moore *et al.* [MRS94]. Their approach relies on a knowledge-based model that maps the functionality of the widgets in the source platform UI toolkit to those of the target platform UI toolkit. Given a particular migration problem, the software engineer identifies the pieces of code in the system implementing the UI. Then, based on the knowledge-based model, a "wizard" guides the software engineer in selecting appropriate widgets in the target platform toolkit that can together deliver the interactive behavior of the original code.

A similar approach was used to address the third category [AFMT95]. The interesting difference is that since there is no source widget toolkit, the reverse engineering process hypothesizes widgets from the code.

Grafting a GUI on top of a relatively non-interactive set of batch or command-line

programs (CLPs), whose functionality is accessible only through command-line inputs, is simpler and does not involve a complex reverse engineering problem. The GUI can be developed independently, with the aid of a GUI-building tool that uses an abstract representation of the underlying applications. Then, the user-initiated events on the GUI are programmed to invoke procedures in the underlying application programs [PA97]. To enable a degree of freedom in utilizing the underlying programs and formatting their outputs, some source code reverse engineering might be required in the beginning to identify internal variables and data structures of interest in order to expose some of them in the new UI [TS99]. Moreover, to allow flexibility in output formatting or to integrate with distributed object middleware, e.g., CORBA, a wrapper can be placed around the command-line application to programmatically invoke its commands. Then, the wrapper parses the generated output and returns a semantically useful result (an integer, an object, etc.) that can be easily consumed by a calling program or a GUI object [WJD01].

Sorzano *et al.* [SCT02] present a model for CLP packages. The model includes a command-line syntax specification, which is integrated into a higher-order OO model that can be directly translated into a graphical user interface. The object types (classes) in this model are: package, group, program, command line, menu and argument, where a package is a CLP application, a group is a subset of related programs in the package and a menu here is a list of arguments. The authors described a language, Colimate (the COmmand LIne MATE) that implements this model. Using Colimate, a GUI description can be written for legacy CLPs and then compiled and run with the help of an interface generator that raises the needed windows, attends to user selections and finally launches the desired processes.

Finally, work in the fifth category, i.e., wrapping with a web-enabled wrapper, is one of few possible solutions to the broader problem of Web-enabling legacy systems, which is discussed in the next section.

Dannelly [D95] presents a case where UI reverse engineering is an objective for itself. He introduced methods for automatic analysis of the source code of X Window System application programs and transforming it into an intermediary representation. This representation is used for automatic production of two types of graphs. The first is widget-instance trees, which are inferred from initialization code and represent the

hierarchy of widgets created by this code. The second is dialog-state diagrams, which are graphical representations of the behavior of X Window System based GUIs and represent the action routines associated with the different widgets created in the initialization code. These graphs improve the system maintainers understanding of the legacy system UI and ease their job.

Except from few examples, the majority of UI reengineering and reverse engineering approaches adopt code analysis and understanding as the means for system modeling and reverse engineering. However, the UI-related code is only part of the system code that has to be examined. Additionally, due to ageing symptoms mentioned earlier, it is hard and expensive to analyze the legacy code. Even worse, in some cases the code is not even available.

In such cases, system-user interaction can be an alternative source of information for understanding the legacy system. It is a rich source of knowledge and a faithful representation of how the system is currently being used. Hence, it is a good candidate input for the UI reengineering process. Interaction reverse engineering in LeNDI is novel and different than the work summarized above in that it uses this input instead of source code for producing the necessary abstractions and models for the consequent forward engineering phase. This reengineering method does not alter or change the existing legacy system code or structure. It relies on the almost-automated process of LeNDI to infer most of the CUI behavior and task models it needs. As described in chapters 5 and six. These models are useful beyond CelLEST and can be deployed in other contexts as described in section 7.3.

2.2 Web-enabling Legacy Systems

Web-enabling legacy systems is the process of opening an existing legacy software for access through the Internet, an extranet and/or an intranet to the public and/or to a selected user-base of employees, customers and/or business partners. The work in this area is mostly industry driven. It is motivated by the emergence of the Internet as a medium for doing business, with new opportunities of business growth and cost reduction. Using the Internet, especially the WWW, a company can reach out to more customers worldwide via a simple, easy to use, inexpensive and very popular UI, namely the web browser [BFM02]. Additionally, via their web sites, businesses can reduce the

cost of transactions and customer service by automating their transactions and moving the effort of customer service to the customers themselves via online orders, quotations, etc.

Since somewhere between 60% and 80% of all corporate information reside on mainframe-based legacy systems [Att00], opening many of these systems for Web-access may be essential for corporate Web-enabling strategies. A number of solutions to this problem were developed mostly by industry, which may require reproducing legacy business data and business processes in new formats and new presentations for old and new audiences [Lan00]. These solutions complete one another in some situations, and compete with one another in other situations. But, in practice a variety of Web-enabling technologies may be used, even within the same corporation. This is due to the wide variety of legacy and Web technologies available and the unique requirements of every Web-enabling project. This section briefly describes these solutions and their advantages, disadvantages and limitations. A legacy software application in general, consists of three layers: the presentation layer, the program-logic layer and the data layer. A Web application can access a legacy system via one or more of these layers, depending on the available legacy and Web technologies and the status of the legacy system.

Increasingly, Web-enabling is becoming one of the main activities in the area of UI reengineering. Web-enabling is also one of the main uses of the CelLEST method for legacy CUI reengineering. Interaction reverse engineering in LeNDI is a significant advance to the practice of modeling existing legacy CUIs for Web-enabling via presentation access. It provides a coherent almost-automated process for modeling legacy CUIs in preparation for Web-enabling that replaces the current manual error-prone time-consuming practices.

2.2.1 Web-enabling via Data Access

In data access Web-enabling, a new Web application is developed to directly access the legacy database and then perform the necessary processing on the retrieved data before presenting it to the user via a web browser. The primary assumption is that, the legacy application logic is trivial and can be easily duplicated in the Web application if required or that new logic will be developed to process data different from the legacy logic. The legacy data must be wrapped in order to be accessed using a different interface or protocol than that for which the data was designed initially. This requires using data

access-middleware in the form of database *gateways* and *bridges***.** A database gateway is a software application that translates between two or more data-access protocols. There are a number of *de facto* industry standard database gateways. Open Database Connectivity (ODBC) is Microsoft's interface for accessing data in a heterogeneous environment of relational and non-relational database management systems. The ODBC API can be invoked inside Active Server Pages (ASP) or programs in C/C++, Perl, VB, etc. Java Database Connectivity (JDBC) is an industry standard defined by Sun for database-independent connectivity between Java applets or applications and a broad range of SQL databases. The JDBC API can be invoked inside a Java program, applet or servlet or a Java Server Page (JSP). ODMG is the standard of the Object Data Management Group for persistent object storage. A bridge is a special gateway that translates one standard protocol into another, e.g., JDBC-ODBC bridge. [CWSR00]

Web-enabling via data access is a simple, straightforward solution [Amb00]. It can provide multi-source legacy data integration to new applications [RMB00]. It has some considerable drawbacks. First, any important logic, e.g., business rules and data validation, is bypassed and not utilized. This means that it has to be re-implemented in the web application. This may cause duplication in program logic and high cost in both development and maintenance. Second, it increases the data coupling between the legacy and web applications [Amb00, RMB00]. This solution is most suitable for Web-enabling legacy services with trivial logic, e.g., Web-enabling the catalog browsing services of a legacy library system. It is also suitable, when new logic needs to be implemented to process the data for Web-access, so, this logic can be implemented in the web application.

Data replication is another data access based Web-enabling method. In this case, part of the central legacy database is duplicated on a web server for Web-access through a client application. The legacy and server data repositories are coordinated with periodic batch jobs. The major weakness of this approach is that it cannot handle real-time data, making it inapplicable for applications such as customer service or sales activity.

2.2.2 Web-enabling via Logic Access

Web-enabling via application logic access relies on the availability of a mechanism to access the business logic independent from the user interface related code. This can be accomplished in different ways:

Code Access. If the legacy business logic is implemented separately from the presentation logic, which is rarely the case, then theoretically, it is possible to insert a thin control layer that accepts the data extracted from the HTML page and invokes the appropriate subroutine from the legacy system. The business-logic subroutine processes the request and gives back the results, which are then forwarded to the client [Sne00].

API Access. Packages like SAP, PeopleSoft, etc., offer APIs that can be accessed via Java Native Interface (JNI) or Common Gateway Interface (CGI) code. But, software developed in-house rarely has a defined API, or at most it has a very limited function-oriented (not OO) API. [Amb00]

Distributed Object Technology (DOT) extends object technology to the net-centric information systems of modern enterprises by using object middleware, e.g., OMG's CORBA and Microsoft COM+. The idea is to *objectify* (or *objectize*) the legacy system by creating an OO interface to individual applications, common services and business data that makes the legacy software look like objects. Then it can be accessed by other applications across a network through the OO interface. [CWSR00, RMB00, ZK99]

This is quite an invasive reengineering solution. The prime challenge is objectifying the legacy system, i.e., analyzing, decomposition, and then translating the monolithic and plain semantics of the usually procedural legacy system to the richly hierarchic and structured semantics of an object-oriented system [CWSR00]. Several methods were suggested to decompose legacy systems into objects, including cluster analysis, concept analysis and hybrid methods [CCDD01, PZKM99]. The amount of effort needed to accomplish this task depends on the language, style and architecture used in developing the legacy application.

Component wrapping is a natural extension to DOT. In contrast with objects, components must conform to a component model. This constraint enables the component framework to provide the component with quality services [CWSR00]. Enterprise JavaBeans (EJB), from Sun Microsystems, is an example of server-side component architecture for writing reusable business logic and portable enterprise applications. EJB is the basis of Sun's Java 2 Platform, Enterprise Edition (J2EE). EJB components are written entirely in Java and run on any EJB compliant server. Each bean encapsulates a piece of business logic. EJB servers provide system-level services such as transactions,

security, threading, state management, resource pooling, distributed naming, remote invocation and persistence. [CWSR00, FOLD96]

EJBs can wrap exiting legacy system functions and offer them as operating system and platform independent components. But like DOT, component wrapping faces the challenge of *componentifying* the legacy application, which is separating the interface of the legacy system into modules consisting of logical units or functions. [CWSR00]

Zou and Kontogiannis [ZK99] describe a combination of DOT and component wrapping technology. First, they identify and generate a decomposition of the legacy system into modules, and then analyze the interfaces of the selected legacy components and store their signatures in a component repository using XML format. Second, they generate the CORBA/IDL and CORBA wrappers from the component repository. Third, they use EJBs to develop the application server, in order to integrate the CORBA wrappers and to provide the services to the Web-based application. Finally, they define a scripting language using XML, to enable the invocation of the legacy components.

2.2.3 Web-enabling via Presentation Access

Web-enabling via presentation access is non-invasive and almost risk-free. It covers a wide spectrum of solutions. At one end, there is Web-enabled emulation of legacy systems. At the other end, there is screen mapping, which allows complex manipulations of the legacy data streams used for communication between the legacy host and the legacy terminals. Thus, it supports crafting reengineered front-ends for legacy CUIs that take advantage of the potential of Web UIs. The common attribute of all these approaches is that the legacy application is accessed via its presentation layer. This can be done mostly by accessing its UI, represented by the legacy outbound and inbound data streams, or in other cases by accessing its presentation description if one exists, e.g., IBM Customer Information Control System (CICS) maps. In both cases, the access is limited to the data and operations offered via the application presentation.

Web-enabling via presentation access started in its simple form of web emulation (webulation [BB01]), shortly after the emergence of the Internet [TLRH98]. Gradually, webulation evolved into the more advanced screen scraping technology that takes advantage of the GUI capabilities of web browsers. The next generation was manual screen mapping that allows remodeling the legacy UI or parts of it into a task-oriented

Web-based GUI. The future trend, which was the subject of our research in CelLEST project, is to automate screen mapping as much as possible, using intelligent tools that can learn about the legacy system on its own. Then, the knowledge learned is used to build the models needed for screen mapping with minimal effort. Consequently, these models are used to build a working Web-based GUI or abstract UI specifications that would allow simultaneous migration to multiple platforms, e.g., the WWW and WAP devices. Additionally, screen mapping is extended to allow the integration of multiple legacy systems UIs together or with other Web applications under a unified Web front-end. In the following we discuss the evolution and available solutions for Web-enabling via presentation access.

2.2.3.1 Web Emulation (Webulation)

Web emulation, or webulation [BB01, BFM02, TLRH98], is a natural extension of the long-practiced legacy host emulation to the Web. The new thing is that the emulator runs in a web browser or a web server. Browser displays have the native look and feel of the host legacy screens. Transactions work exactly the same as on a legacy host "green screen" terminal, e.g., IBM 3270, by returning one screen display for one input request. Full support for legacy function keys as well as user customization of colors and fonts are usually available. Additionally, icons for function keys, copy/paste, macro recording, file transfer and other basic operations are provided. [Ake00, BFM02]

Webulation is a quick and cheap solution that does not need any Web application development. It offers instant access to the legacy application to intranet and extranet users who are already familiar with the legacy system. But, for the wide Web population or new users, it does not make the legacy system any easier to use [BFM02]. Additionally, it does not allow tailored Web-access that targets different groups of users with different limited sets of UI functionality. It addresses the accessibility issue, but it does not improve the usability or navigability of the legacy system (see section 1.2). A typical implementation of webulation is done using Java applets, downloaded into the client web browser. The applet runs in the web browser Java runtime environment and establishes a connection with a Telnet server that manages access to the host application. [BFM02, TLRH00]

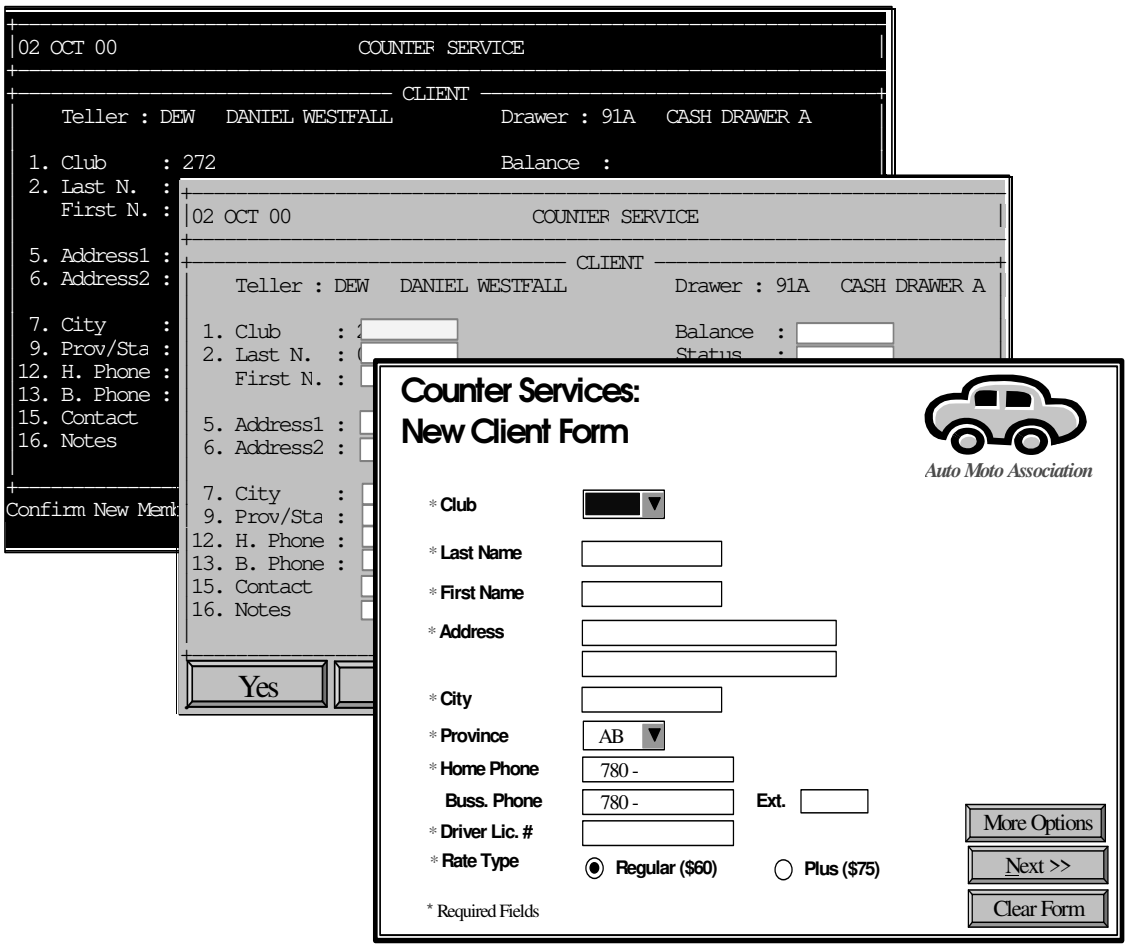

Figure 2.1. An Example Legacy Screen (upper), Refaced On-the-fly (middle) and Refaced Using Screen Customization (lower)

2.2.3.2 Screen Scraping (Refacing)

Screen scraping (or refacing [Ake00, Att00]) takes webulation a step further by offering an enhanced *one-for-one* browser presentation of the legacy UI. It reads the data stream intended for the mainframe terminal, either via a client based terminal emulator (Java applet) or a server based program, and turns it into a Web-based GUI presentation. The translation of each legacy screen to a Web-based GUI can be done in two ways, either on the fly or using a user defined customization for this screen. Figure 2.1 shows both cases.

In the first case, a middleware is interleaved between the Web front-end and the legacy software to act as a presentation translator by intercepting outbound legacy displays and converting them "on-the-fly" into a Web front-end using whatever available

information. In case of IBM 3270 for example, this is done by converting the unprotected (input) fields into edit control objects while turning the other fields into labels. Some advanced tools convert a set of predefined strings like F1,… F24 into buttons. Slight improvement is achieved over webulation.

In the second case, an individual customization is created for every screen that radically changes its appearance and takes advantage of the presentation potential of the target platform, the Web in this case. Thus legacy screens appear "dressed up" in a Web-based GUI, with widgets, lists, radio and push buttons, images, web links, check and choice boxes, colors, fonts, etc. Additionally, one can reorder fields, change tab sequence, and hide unnecessary data [Ake00, BB01, BFM02]. Figure 2.1 shows a legacy screen that is refaced on-the-fly (middle) with unprotected (input) fields turned into text boxes and "confirm new member ? (Y/N)" message replaced by "Yes" and "No" buttons. The same screen is refaced with individual customization that turned the original screen fields into text boxes, lists, radio buttons, etc., wherever suitable. Also, a logo and a big font title were added. A few buttons with additional functions were added.

Some screen scraping tools, e.g., IBM Screen Customizer [IBM99, BFM02], offer the ability to create context-sensitive field help for host applications, to create a list of valid values for a data field and/or to skip unnecessary screens during navigation. Also, they allow customization templates to be applied, in order to speedup refacing a number of legacy screens. Such a template would contain common elements to all Web-based screens replacing the legacy screens, e.g., a logo, a web link, a customized tool bar, etc. The operation of the applications is still "one-for-one". That is, one browser request equals one legacy screen display [Ake00].

To know which customization to apply to a screen snapshot, the screen scraping application should recognize the identity of the instance. This is done using a predicate or signature for every screen. Typically, this signature is based on some unique keyword(s) that appear on the screen at some location(s). Some tools offer rich pattern definition languages for the application builder to define a signature [BFM02, Cel99]. Such languages would allow specifying that a specific text must exist or not exist at a certain location or within an area on the screen and/or that it must be of a certain case or color, or compares in a certain way to a hard-coded value (<, <=, =, >=, >). They may also allow

multiple recognition criteria to be defined and combined with logical operators, for the same screen.

In many cases, this technique would be sufficient to recognize the identity of the screen. However, in some cases, e.g., unstructured and multi-mode screens, it is quite challenging to find such a signature. For example, some host applications can have more than one mode for the same screen, e.g., Create, Review, or Update modes, with the same structure and appearance but with slight differences in the status of some fields. Each mode needs separate signature and customization. In the state-of-the-art practices, a screen signature must be manually defined and hard coded for every screen by an expert analyst, who is very familiar with the legacy system under analysis and with the pattern language available and its supporting tool.

Screen scraping takes relative advantage of the presentation capabilities of the web browser. However, it does not benefit from its enhanced navigability. So it enhances the accessibility and, to some extent, the usability but not the navigability of the legacy system.

Another screen scraping approach is to access the legacy presentation at a level lower than the UI, or the data streams used to construct it. This is the level of screens maps or description files, if such concept exists in the system under study, e.g., CICS maps for S/390 systems and Data Description Specifications (DDS) source files for AS/400 systems. In case of CICS, the data necessary to build an HTML or XML UI is extracted from CICS maps instead of IBM 3270 data streams.

2.2.3.3 Screen Mapping (Remodeling)

Screen mapping [Cri01], also called remodeling [Ake00], is a natural extension to screen scraping. While it still uses presentation access to Web-enable legacy systems, it allows reengineering the legacy UI or chosen parts of it into a task-oriented Web-based GUI. It enables fairly extensive modifications to the sequence of information presented to the user by combining several screens into a single graphical presentation, i.e., it offers *many-for-one* browser presentation of the legacy user interface [Att00]. Thus, the multiple host screens, related to a certain user task are combined in one (or more) Web forms that is a more natural representation of the task in the Web world. This can greatly enhance the usability and navigability of the system, while still maintaining the back-end

host navigational flow. It is possible to partially apply screen mapping to a legacy system by reengineering the frequent or cumbersome user tasks of choice, while using webulation and/or screen scraping for the rest of the system.

To build a screen mapping solution, one needs to do the following:

1. Build a model of the portion of the legacy system CUI to be reengineered,
2. Describe the steps needed in terms of user actions, inputs and outputs, and screens accessed to perform each user task that will be reengineered,
3. Build/buy the middleware needed to mediate between the legacy back-end and the Web front-end, and
4. Build a Web-based GUI for each user task, which will be responsible of executing the task plan via the host-access middleware.

The partial model built for the legacy CUI is a set of predicates or signatures that should uniquely identify each legacy screen, along with a list of the possible behaviors of each screen in terms of the user actions permissible on it and their outcomes or destination screens. This model is like a road map for the legacy CUI.

A task description gives the detailed steps of how to open sessions, gather data, complete transactions, and close sessions with the host legacy to accomplish a user task. Typically, this includes what user actions are needed to navigate the legacy CUI in service of the user task, what inputs need to be passed to the legacy application on which screens on which locations and what outputs will be retrieved from which locations on which legacy screens.

A host-access middleware executes the task description by driving the necessary navigation via the legacy host, passing the user inputs received from the Web-based GUI to the legacy application, and collecting the required outputs to feed the Web-based front-end. This middleware uses terminal access protocols such as VT100, IBM 3270 or IBM 5250 to communicate with the legacy system via a “virtual terminal”, emulating the standard “green screen” terminal. Data are moved in and out of the legacy host via the legacy system CUI as if data entry personnel were flawlessly entering it [Cri01]. Such middleware can be built with EJB beans, Java servlets, or similar approaches.

The Web-based GUI, e.g., HTML or XHTML, presents the reengineered UI to the user, takes his inputs, executes the task plan and reproduces the collected results through

the web browser. Common Gateway Interface (CGI) scripts, Java Server Pages, or similar technologies are used to collect the user input via the client web browser

In current practices, these steps are all implemented manually. An analyst goes through each screen of the legacy system, trying to find a unique signature for it and to model its behavior in order to build the legacy CUI model. Then s/he talks to the users about every task to be migrated to the Web to understand all its possible paths and exceptions. Then, s/he manually builds a plan of this task, by describing all the user actions needed to perform it, all the inputs to be entered and where they occur and all the outputs to be collected and from where they are obtained. After that, a developer builds the new Web front-end that executes the user tasks. For each task, s/he does the necessary GUI design, layout and coding. If it is required to migrate to a different platform too, e.g., WAP devices, then the legacy CUI model and task descriptions can be reused but the new UI implementation should be carried out from scratch for the new platform.

2.3.4 Pros and Cons of Web-enabling via Presentation Access

A market survey [Att00] showed that 60% of the IT personnel administrating, maintaining or accessing information from legacy systems use some form of screen scraping or screen mapping technology to integrate legacy systems with other systems. 45% of them batch data to a server for access through a client application, and a similar number modify the host application to suit client access. The study showed that out of those who use presentation access technology, 60% use it to avoid changing the host application, 30% could not change the host application and 44% use it for its lower cost.

Despite the possible bias in market studies, these results summarize the advantages of Web-enabling via presentation access. It is a minimal-risk, non-invasive, less expensive solution. It requires no change to the legacy application. This makes it almost the only choice when changing the legacy application is not an option, e.g., due to lack of ownership or unavailability of the source code. It can be applied gradually and/or using a mixture of methods. For example, webulation can give instant Web-access of the legacy application to the users familiar with it, while a screen mapping solution is deployed to reengineer the UI of the most frequent/difficult user tasks. This provides an easy to use HTML front-end to the external users with no familiarity with the legacy system, e.g., customers placing orders and college students registering for classes. Finally,

presentation access can be used for lightweight integration with other legacy systems and Web applications or for limited functionality extension and repurposing of the legacy system.

On the other hand, screen scraping and screen mapping are labor intensive non-automated processes. The currently available tools for supporting them are mostly limited to aiding the manual practices. They do not automate any of the subtasks involved, which may require a lot of effort and intuition. Additionally, these technologies are better suited to mature stable applications, which are unlikely to go through frequent changes or updates. For dynamic applications that go through frequent changes, keeping the Web application layer up to date with the latest changes incurs high maintenance overhead.

Presentation access of legacy systems is criticized for being slow, since it adds a remote extra layer on top of the existing legacy application. This depends on the implementation model used. Modern server side and host side implementations can overcome this deficiency to a good extent. In a host side implementation, the host-access and the screen mapping middleware reside on the legacy host, e.g., S/390. A user task is executed completely on the host and HTML pages are generated as needed and submitted to the user with the required results or to collect inputs. However, fair comparison with the other Web-enabling approaches presented earlier is unavailable to judge their relative speed and scaling up with workload.

Another disadvantage of presentation access solutions is their vulnerability to unexpected events related to the host connection behavior, like keyboard lockups, session disconnections, broadcast messages from hosts and error messages coming from the legacy application [Yam00]. Careful analysis and modeling of the legacy CUI and the tasks to be reengineered can reduce this risk by anticipating as many of such events as possible and including a recovery mechanism in the Web front-end application, but would require more investment and effort.

Finally, Web-enabling via presentation access has some limitation: it cannot extend the legacy system functionality beyond what is already achievable, directly or indirectly, through the legacy presentation. It only gives access to the data exposed through the legacy presentation.

The CelLEST project adopts presentation access for legacy CUI reengineering and Web-enabling. LeNDI overcomes some of the disadvantages of presentation access solutions by providing a coherent almost-automated process and tool support for interaction reverse engineering and legacy CUI modeling. LeNDI uses traces of interaction with the legacy CUI as input. LeNDI provides a data mining method that discovers the frequent user tasks of interest in the form of interaction patterns, as they are evident in the interaction traces. These patterns are used to build the task models that will be encapsulated in the reengineered UI. LeNDI reduces drastically the level of skills, time and cost needed for the modeling process, since its subtasks are almost automated. Additionally, the automated process is less sensitive to changes in the underlying CUI as changes can be captured by recording and analyzing new or extra traces instead of implementing them manually. LeNDI advances the current screen mapping practices and lays the foundation for the future generation of these solutions.

2.3.5 Objectifying Legacy Systems via Presentation Access

Little work has been done along this somewhat different line, which mixes presentation access of legacy systems with distributed object technology. There is very little research in this area, and it suggests screen scraping of legacy CUIs, or similar techniques for legacy GUIs, as a means for wrapping a legacy system service (or a user task) as a method in an object for consumption by a new application or in a distributed object environment.

Chadha [Cha98] describes a prototypical effort, during which, the services of several back-end legacy systems belonging to various health insurance providers were integrated in one distributed object environment to provide access to these systems to healthcare providers. This was done via a distributed object, called the “Payer” object. Each Payer object wraps a legacy data source, e.g., an IBM mainframe application, an ODBC-enabled database or others. The Payer object interface allows healthcare providers to check the eligibility of patients for insurance coverage, submit insurance claims, and check the status of submitted claims. The object interface is described using CORBA IDL. For IBM mainframe-based legacy back-ends, the Payer object uses screen scraping to invoke the procedures that perform the required services on the legacy application and collect the required results.

A similar approach was followed to objectify legacy GUI-driven applications (GDAs) and integrate them with other GDAs, including Web applications, to form mega-applications [GBP02]. This requires the use of agent processes to be injected into the GDAs. These agents collect information on all GUI elements that are used by a GDA, monitor events that are generated, and trigger GUI input events. An agent presents an object (representing an objectified GDA GUI) to a controlling program. This program can then invoke methods on specific GDA GUI elements and replay GUI inputs with the support of the agent. This controlling program can integrate a number of GDA GUIs together, including Web-based GUIs.

2.3 Software Requirements and Process Model Recovery

Requirements recovery is the process of retrieving software functional and user requirements and/or software specifications from an existing software, its documentation, its stakeholders and its operation environment. Requirements recovery research is fairly scarce. Previous work in this area had explored a variety of methods that assume different input information and recover various different types of requirements. In LeNDI, we developed interest in requirements recovery research because part of LeNDI's role in CelLEST process is to discover the system services that may be candidates for reengineering and wrapping with a new front-end. Generating hypotheses about the system services, as they are used today by current system users, in the form of interaction patterns, is essentially a requirements-recovery activity. So, it was important to review the related literature. Despite the variety of interesting approaches that were used for requirements recovery, none of them used interaction traces as input. Hence, LeNDI needed a novel method for interaction pattern discovery from interaction traces as described in chapter 6.

In the REVERE project [REGS00] probabilistic natural language processing (NLP) methods were employed to recover software requirements from the available documentation, such as requirements specifications, operating manuals, user interview transcripts and data models. The method suffers from the well-known shortcomings of NLP and needs to be adapted (trained) to the various documentation styles, structures and notations, but provides rapid analysis for voluminous documentation.

Cohen [Coh94] used Inductive Logic Programming (ILP) to discover specifications from C code. The constructed specifications are in Datalog (Prolog with no function symbols). The software discovered two thirds of the specifications with about 60% accuracy, in a program containing over one million lines of source code. The system recovered declarative view specifications from relational database examples. Positive examples were obtained from program execution views, with background knowledge, consisting of the table relation. The technique uses inductive reasoning about the behavior of the code, rather than deductive reasoning of static code. Sufficient training data is required, otherwise results will contain numerous inconsistent specifications.

The AMBOLS project [LAQ99] aimed to recover requirements by employing semiotic methods and intensive interviews with the stakeholders to analyze and model the system behavior from various viewpoints. The intent is to document current uses for the purpose of redeveloping the application.

In [SP99], data reverse engineering was proposed as a means for business rules recovery from legacy information systems. Particularly, an approach for extracting constraint-type business rules from database applications was outlined, but without an implementation or experimental evaluation.

Di Lucca *et al* [DFD00] presented a method for recovering a use case model from threads of execution of object-oriented (OO) code. A thread is a sequence of method executions linked by the messages exchanged between objects. Threads are triggered by input events and terminated by output events. In this approach, developers identify statements that form input events and output events. A tool then automatically identifies the code corresponding to the potential uses cases. The tool produces a structured use case model including diagrams at various levels of abstraction, comprising actors, use cases, associations between actors and use cases, and relationships among use cases. The mapping between a given use case to its corresponding code supports developers in program understanding and maintenance impact analysis. The method targets OO systems, which makes it inapplicable to most legacy systems that were developed before the wide spread of the OO paradigm.

Similar to this research area, is the work on process model discovery, e.g. [AGL98]. The idea is to model existing known or unknown processes by mining workflow and

other logs of these processes and retrieving or discovering the models. The aim is to gain better understanding of the existing process models and/or develop future ones. An example of this work is reverse engineering work processes in collaborative virtual environments [BS02]. The goal was to recover design models of virtual workspaces at micro (individual tasks), macro (processes) and collaboration (task sequences) levels, by mining the environment's data logs, e.g. threads of email messages posted on the bulletin board and actions performed by the collaborating team members.

The work presented above represents diverse directions in exploring and tackling the requirements recovery problem. Researchers explored different available inputs, e.g., existing documentation, human knowledge, code, data, threads of OO program runs and workflow logs. In this thesis, we introduce a new method for recovering the *de facto* functional requirements of legacy systems to support the CelLEST method for legacy CUI reengineering and Web-enabling. To do so, we employ another yet unexplored easy-to-collect input, which is records of the system-user dialog via the legacy CUI in the form of interaction traces. LeNDI applies data mining algorithms to these traces to recover the needed requirements in the form of interaction patterns. There are a number of potential uses of this promising approach beyond the forward engineering phase of CelLEST, as suggested in section 7.3.

2.4 Sequential Data Mining

Mining sequences of data for recurring patterns is a generic problem with instances in a range of domains. It was first introduced in [AS95] under the name "sequential pattern mining", inspired by applications in the retail industry. Given a set of customers and their sequences of transactions, the goal is to discover sequences of items (patterns) occurring in the transactions of the same customer.

In CelLEST, it is necessary to discover the frequent legacy CUI navigation sequences that represent multiple uses of the same legacy system service, from a system viewpoint, or repetitive executions of the same user task, from a user viewpoint. LeNDI mines the recorded traces of interaction with a legacy system for these interaction patterns. This problem, called interaction pattern mining problem, is different from "mining sequential patterns", but similar to the problem of "discovery of frequent episodes in event sequences" [MTV97]. In [MTV97], the discovered frequent episodes or patterns can have

different types of ordering: full (serial episodes), none (parallel episodes) or partial and have to appear within a user-defined time window. The support of a pattern is measured as the percentage of windows containing it. Some algorithms were developed to tackle this problem, e.g. WINEPI and MINEPI [MTV97] and Seq-Ready&Go [BCB00], based on the famous data mining Apriori algorithm [AIS93, AS94]. Apriori was originally proposed to solve the problem of mining association rules between sets of items in large databases and then numerously extended to solve other data mining problems including mining of sequential patterns. The problem of mining interaction patterns differs than the formulation of [MTV97] in that it does not restrict the pattern length with a window length and permits a user-defined number of insertion errors to exist in the instances of the discovered patterns.

The CelLEST interaction pattern discovery problem is also similar to the problem of discovering patterns in DNA and protein sequences. There, the objective is to discover either probabilistic patterns or deterministic patterns with noise, e.g. flexible gaps, wild-cards (don't care characters) and/or ambiguous characters (which can be replaced by any character of a subset of the alphabet set, A) [BDVHHP00]. Because bio-sequential data is usually very large, an efficient search strategy is to discover short or less ambiguous patterns using exhaustive search, possibly with pruning. Then the patterns that have enough support are extended to form longer or more ambiguous patterns. This process continues until no more patterns can be discovered. Two elegant examples of this category are PRATT [Jon96] and TEIRESIAS [Flo99] algorithms. PRATT can discover patterns of the quite general PROSITE format [BB94], e.g. C-x(5)-G-x(2,4)-H-[BD], where B,C, D, G and H $\in A$, x(5) is a flexible gap of length 0 to 5, x(2,4) is a flexible gap of length 2 to 4, and [BD] is an ambiguous character that can be replaced by B or D. The original TEIRESIAS algorithm discovers $\langle L,W \rangle$ patterns with wild-cards only, where $L \leq W$. An $\langle L,W \rangle$ pattern has a constraint on its density, that is any of its sub-patterns containing exactly L non-wildcards items has length of at most W items. For example, CD..CH..E is a $\langle 3,5 \rangle$ pattern, where '.' can be replaced by one item $\in A$. None of these two bio-pattern discovery algorithms mentioned above suits the task of mining interaction traces for interaction patterns.

None of the problem formulations described above matched the needs of LeNDI, especially in terms of the type of ambiguity or errors that they allow in the patterns discovered. LeNDI discovers patterns of user activity in the traces of interaction with a legacy CUI with insertion errors, i.e., whose instances may contain up to a user-defined number of spurious activities. A spurious activity happens when the user accesses or receives a screen that is not part of the task s/he is performing, e.g., an error or help screen. LeNDI treats such activities as insertion errors and allows up to a pre-set number of them to exist anywhere in a pattern instance. Two novel pattern mining algorithms were developed, specifically to solve the interaction pattern mining problem in LeNDI: Interaction Pattern Miner (IPM) and Interaction Pattern Miner 2 (IPM2). The first is a breadth first algorithm and the second is a depth first algorithm. Both algorithms require defining a criterion for pattern selection and use the idea of building longer patterns from shorter ones. Although designed for use in LeNDI, they can be applied to similar problems. In fact we used them to mine user web site navigation logs for interesting navigation patterns [ES03]. Both algorithms are described, compared and evaluated in chapter 6.

Chapter Three

CelLEST User Interface Reengineering

In chapter 2, various legacy CUI reengineering, reverse engineering and Web-enabling methods were introduced. It was shown that almost all CUI reengineering and reverse engineering methods rely on code analysis and understanding, except some methods that deal with command-line programs. This makes such methods inapplicable when the code is unavailable or unchangeable. It also makes them costly and risky when code is hard to comprehend and difficult to change. Nevertheless, in some cases, it is unavoidable to do UI reengineering via code analysis and change since the code will be migrated or for other reasons. This raises the need for new CUI reverse and forward engineering methods that do not use code in order to serve cases when code change is impossible or undesirable. This thesis proposes a novel lightweight CUI reverse engineering process that utilizes, as input, traces of interaction with legacy CUIs. It is a valuable method when code and platform migration is not necessary and lightweight CUI reengineering will be used.

Additionally, chapter 2 presented the different strategies of Web-enabling legacy systems. It showed the reasons of popularity and advantages of Web-enabling via presentation access. These reasons are non-invasiveness, low risk, lightweight and low cost. It also discussed the shortcomings of this method, which are labor intensiveness, manual processes, inadequacy of tool support, vulnerability to unexpected events and to changes in the legacy CUI and inability to extend the system functionality significantly. The interaction reverse engineering method proposed in this work and implemented in LeNDI overcomes some of these shortcomings by proposing a coherent lightweight reverse engineering method for legacy CUIs in service of Web-enabling via presentation access. Unlike current manual industrial practices, this method is almost-automated, less error-prone, more productive and less sensitive to changes.

Since, a significant part of this thesis was done within CelLEST project for legacy CUI reengineering, it is important to describe this project first before describing the specifics of our interaction reverse engineering method. CelLEST project [SEKSM99,

SES02, SESP00] is a joint research project between the Software Engineering Lab. at University of Alberta, Canada and an industrial sponsor, Celcorp [Cel]. The aim of the CelLEST project is to develop the next generation of legacy CUI reengineering and Web-enabling via presentation access, using artificial intelligence (AI) and other methods. CelLEST adopts interaction reengineering as a means to automate the process of "learning" and reengineering an existing legacy CUI. CelLEST uses a combination of document analysis, feature extraction, clustering, user action modeling, visualization, data mining, task model inference, XML wrapping and automated GUI layout to develop an intelligent semi-automated lightweight method and prototypes for legacy system CUI reengineering, Web-enabling and front-end integration. The CelLEST CUI reengineering is a two-phase process; the first is a reverse engineering phase and the second is a forward engineering phase. The overall CelLEST process is shown in Figure 3.1.

In the reverse engineering phase, the users' interaction with the legacy system CUI is recorded using a specially instrumented emulator. The recorded traces consist of the screen snapshots accessed by the users while navigating the legacy CUI, the actions they performed on these screen snapshots and the sequences they followed during their navigation to accomplish their work. Then, these traces are used to build a behavioral state-transition model for the legacy CUI (Task T1). This model is a road map for the legacy CUI. It is used by the new reengineered UI to verify the identity of legacy screen snapshots while they are accessed to perform a user task, and hence input the appropriate inputs and deduce the required outputs. Additionally, data mining algorithms are applied to the interaction traces to discover frequent patterns of interaction with the legacy system (Task T2). Each pattern is enriched with additional semantic information to build a model of the corresponding system service or user task, in terms of the interface navigation and the information exchange it implies (Task T3).

In the forward engineering phase of CelLEST, the user task models are translated in to abstract GUI specifications in XML (Task T4). These specifications are then translated to XHTML for Web access or WML (Wireless Markup Language) for WAP (Wireless Application Protocol) access, using the appropriate CelLEST interpreter (Task T5). The strength of this approach is in that it can accomplish simultaneous reengineering of the

same legacy CUI to suit different platforms, using the abstract GUI specifications, which are platform independent.

The CelLEST method for semi-automated "learning", modeling, reengineering and Web-enabling of legacy CUIs and the user tasks of interest makes it much easier to deal

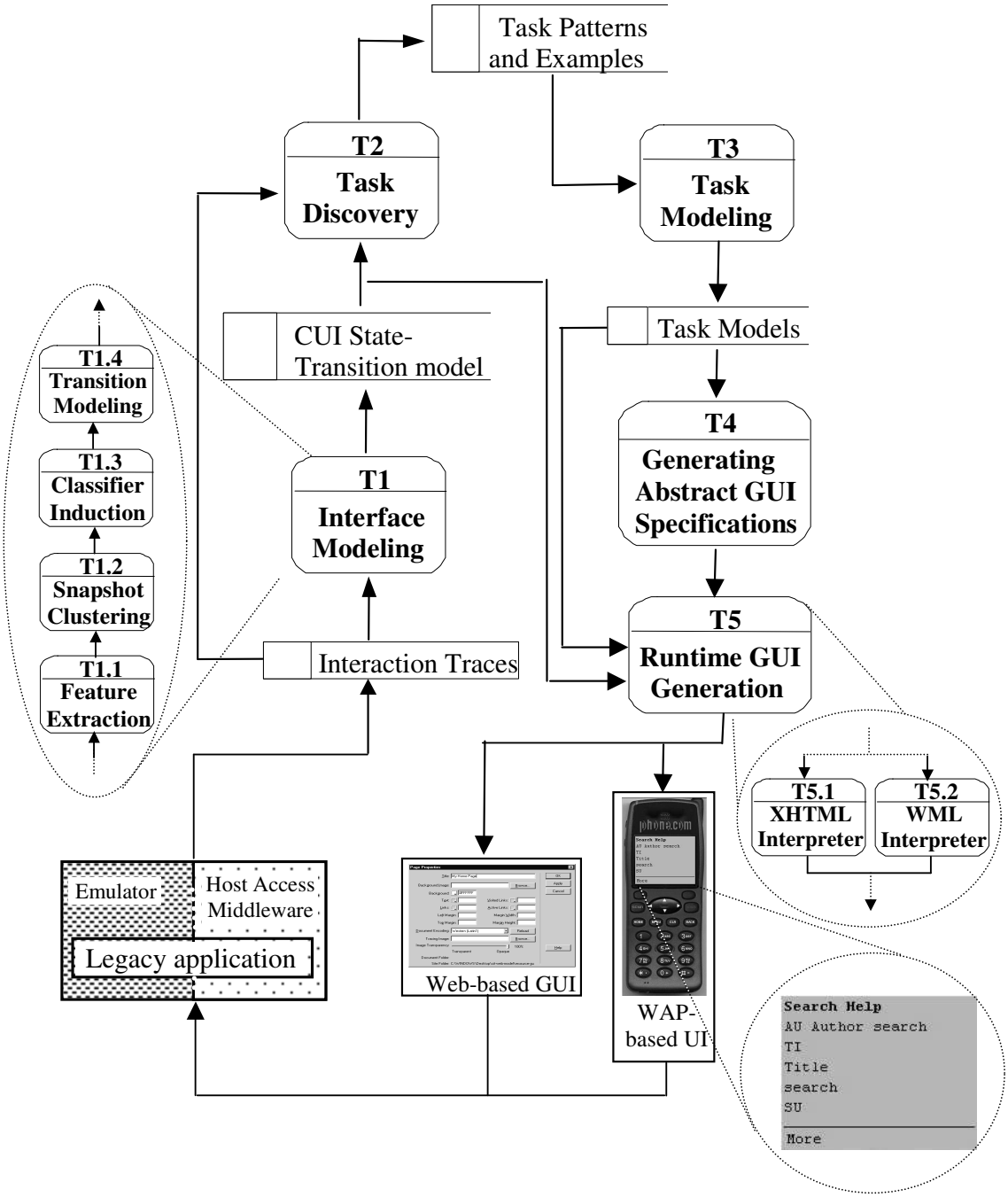

Figure 3.1. CelLEST User Interface Reengineering Process.

with frequently changing legacy systems, since manual re-modeling and GUI re-development may be quite costly. The following sections brief the CelLEST process and show its advantages over the current practices. In Figure 3.1, Tasks T1, T2 and T3 are the reverse engineering phase of the process. T1 and T2 were the subject of this work and are described in full details in this thesis. T4 and T5 are the forward engineering phase, and specifically T5 is a runtime task. T3, T4 and T5 were developed by other CelLEST project members and their different versions are described in detail in [Kap01, Kon00].

3.1 Interaction Traces Collection

While the users of the legacy system are performing their regular tasks, their interaction (dialog) with the legacy UI is recorded in the form of traces or sequences, using a specially instrumented emulator. For block-mode data transfer protocols like IBM 3270, a trace is a collection of screen snapshots forwarded by the legacy application to the user's terminal, interleaved with user actions in the form of sequences of keystrokes performed in response to receiving a screen snapshot. Additionally captured information, in case of IBM 3270 data streams, include the total number of fields, the number of unprotected fields and the initial cursor position. We call these recorded traces "interaction traces". Formally, an interaction trace is defined as follows:

Definition 3.1

$Trace_{id,n} = snap_{id,1}\ (key_{id,j}\ snap_{id,,j})^*\ j = 2...n$, where

- *id* is the trace Id,
- *n* is the length of the trace,
- *j* is the j^{th} screen snapshot received at the user's terminal, and
- key_j is the sequence of key-presses issued by the user at snapshot $snap_{j-1}$ that caused the application to send the next snapshot $snap_j$ to the user's terminal.

3.2 T1: Legacy Interface Behavior Modeling.

The purpose of this task is to build a behavioral model for the legacy CUI, in the form of a state-transition model [EISSM01, SEIS03, SEKSM99]. Each node (state) of the model corresponds to a screen of the legacy system CUI, identified by a unique predicate. Using automatically extracted features for every screen snapshot recorded in the traces, a clustering algorithm groups similar snapshot together, as instances of the same screen. Then a classifier induction algorithm is applied to the snapshots of the identified clusters to automatically infer a unique predicate for their screen, i.e., for the corresponding node

or state on the state-transition graph. Using a pattern-inference algorithm, the arcs of the state-transition model are inferred automatically. Each arc models a permissible user action. The details of the algorithms used in this task are given in chapters 4 and 5. The produced state-transition model is road map for the legacy CUI. It allows a new reengineered UI to check the identity of incoming snapshots while accessing them online against the nodes or states of the model, and hence input the appropriate inputs and extract the required outputs relevant to the executed user task. Additionally, the state-transition model can be queried about navigation paths from a state to another, thus helping planning new tasks that are achievable through the CUI although not originally intended by system developers. The LeNDI (***Le****gacy* ***N****avigation* ***D****omain* ***I****dentifier*) prototype was developed to test the methods and algorithms used in this task. LeNDI deals with data transfer protocols that are native block-mode protocols or can be emulated in block-mode, e.g., IBM 3270 and VT100. In the sequel, we brief the subtasks of task T1 in Figure 3.1.

3.2.1 T1.1: Feature Extraction

In order to automatically build a legacy screen classifier that is able to distinguish the snapshots of each screen using unique screen signatures or predicates, one needs a rich set of features. LeNDI employs a variety of document analysis techniques to extract visual and other features for every snapshot. The output of this subtask is a feature vector for every snapshot. These features include:

- The existence of special system keywords, sentences or information at the top or bottom of the snapshot, e.g., title, code, date, time or page number.
- The information received with the outbound legacy data streams, e.g., the location and type (protected or unprotected) of IBM 3270 data fields and the cursor position.
- Snapshot layout features like the classification of the snapshot to “general”, “table” or “list” with some attributes for the last two classifications, e.g., the number of columns, rows, etc. Another layout feature is vertical and horizontal histograms built for the entire snapshot content or only for some special characters of interest, e.g., numbers.

LeNDI employs 14 single-part and multi-part features. Additionally, it has a binary[1] feature suite of 39 features, constructed by decomposing and abstracting these 14 features. This binary feature suite is needed for one of LeNDI's clustering algorithms. Chapter 4 describes in detail LeNDI's feature suite and all the algorithms used to extract them, along with the similarity measuring metric used for each feature.

3.2.2 T1.2: Snapshot Clustering

Snapshot clustering is the process of grouping similar snapshots together to infer their common identity, represented by a signature or a predicate that uniquely distinguishes them from other snapshots. LeNDI employs two clustering techniques. The first is a single-path incremental clustering algorithm [SEKSM99]. The second is a top-down clustering algorithm [EISSM01]. The incremental algorithm goes over the snapshot set only once, accessing it one by one. Using a user-defined similarity function, the algorithm assigns a new snapshot to the most similar cluster of the clusters available so far, or assigns it in a new cluster if it is not similar enough to any existing cluster. The algorithm requires the user to set up a similarity function and the similarity threshold that decides if a new snapshot is to be placed in a new cluster or to join an existing one. In addition, it requires sufficient familiarity with the system in hand. In return, it does not need an estimate of the number of clusters sought. The top-down algorithm uses LeNDI's binary feature set. It requires as input an estimate of the number of clusters and does not assume familiarity with the legacy system under analysis. Initially, it assigns all the snapshots in a single cluster and then keeps splitting clusters iteratively until reaching the desired number. In each iteration, the algorithm employs an internal cluster incoherence measure to split the most incoherent cluster using the feature value that minimizes the maximum incoherence of the clusters resulting from the split. This algorithm produces a decision tree that reflects the hierarchy of the splitting decisions used to produce the resulting clusters.

The user can choose which algorithm to use depending on the system under analysis. LeNDI's clustering process is interactive. The LeNDI analyst performs few rounds of clustering with different setups to enhance the obtained results. When reaching a satisfactory clustering of the data set, s/he can review and correct the results by moving

[1] Binary here means a feature whose comparison yields only either one or zero.

misclustered snapshots to their right clusters and joining redundant clusters with their originals.

3.2.3 T1.3: Classifier Induction

Given the snapshot clusters resulting from subtask T1.2, a classifier can be induced that can correctly classify individual snapshots into their corresponding clusters. This classifier can then be used at runtime to recognize new, previously unseen snapshots as instances of the CUI screens, and hence, to infer what actions are possible on each snapshot and to which screens they lead. In addition, verifying the snapshot identity allows the new reengineered GUI to apply whatever relevant input or output steps of a task plan to the snapshot, via the host-access middleware.

LeNDI employs two classifier induction algorithms. The first is a signature-based classifier that is induced by superimposing the snapshots of each cluster and capturing what is common in their feature vectors and presentation spaces. The second is a decision tree classifier, which is associated with the top-down clustering algorithm. It is induced by applying the user feedback for fixing the results of the top-down clustering algorithm to the decision tree produced by the algorithm. The fixed decision tree classifier is then used at runtime to infer the identity of new snapshots. The classifier produced should be used to classify new data to test its ability to generalize its knowledge and its accuracy.

3.2.4 T1.4: Transition Modeling

Transition modeling is the process of inferring a model for the transition needed to transfer the legacy system CUI from a screen to another, i.e., from a state on the state-transition model to another. Such a model includes the origin screen Id, the destination screen Id and a model of the user action needed to do the transition. Different styles of user-interaction with legacy systems exist, e.g., function keys, menu-driven, command-driven, and form-filling. Also, an action can have several formats; e.g., a command keyword may have multiple synonyms or it may have an equivalent function key. Currently, LeNDI can model command-driven and function keys styles. LeNDI infers each action model by comparing the instances of this action recorded in the interaction traces and applying a set of rules for command language design. For each action, LeNDI infers its syntax in terms of the function or control key(s) used and the command keyword(s), its options and its arguments. For the arguments, it infers their number, their

syntax if any and whether they are optional or mandatory. The more instances are available, the more general and accurate the action model is. The LeNDI analyst can override and rewrite or fix an inferred action model.

3.3 T2 and T3: Frequent User Task Discovery and Modeling

The purpose of these two tasks is to automate the process of modeling the frequent user tasks of interest as much as possible. Hence, T2 and T3 save the labor-intensive work needed to define all the possible navigation paths of every task that needs to be reengineered and every piece of data exchange that takes place during the task. This is done by automatic learning from the interaction traces about the frequent user tasks in terms of what navigational path is traversed and what type of input is entered on which location on which screen for every task. To know what output is of interest to the user during a task, i.e., what information on which screen is retrieved, the analyst and/or an expert user need to manually identify this information on the snapshots of some instances of this task. This is because these outputs are visually retrieved by the legacy system user, i.e.; s/he just reads them on the screen or prints them. However, s/he does not take any action that can be recorded in the traces as evidences of which areas on the screen display these outputs.

3.3.1 T2: Task Discovery

LeNDI automates the discovery of users' frequent interaction patterns with the legacy system, which represent frequent uses of the legacy system services or frequent executions of the important user tasks. Two algorithms for sequential pattern mining were developed in LeNDI for this purpose, called Interaction Pattern Miner (IPM) [ESS02b] and Interaction Pattern Miner 2 (IPM2) [ESS02c]. Both algorithms can discover similar segments of interaction with the legacy system, in the recorded traces, even with some noise in the form of spurious irrelevant screens. Accommodating noise gives LeNDI flexibility in discovering tasks that include user mistakes, unnecessary navigation like invoking help screens and/or alternative paths for some subtasks. IPM is a depth-first algorithm, while IPM2 is a breadth-first algorithm. They require defining a criterion for interesting patterns in order to use it for deciding if a pattern is worthy of reporting or not. The criterion includes the pattern's minimum length, minimum number

of occurrences, minimum score and the maximum number of insertion errors allowed in any instance of this pattern. The scoring function used is explained in chapter 6.

After reviewing the discovered patterns, the analyst needs to decide whether each one of them corresponds fully or partially to a real frequent user task, or is just a spurious repetition of a navigational path. The instances of each user task can then be used to build the corresponding task model.

3.3.2 T3: Task Modeling

Mathaino [Kap01, KS01, SK02] is another prototype tool of CelLEST. It accomplishes a reverse engineering task, T3, and two forward engineering tasks, T4 and T5. Mathaino replaced an earlier CelLEST tool URGenT (User interface ReGENeration Tool) [KSM99, SEKSM99, Kon00]. Mathaino generalized some of the concepts developed in URGenT by allowing more flexibility in defining task models and by supporting simultaneous legacy CUI migration to multiple platforms using intermediate platform-independent GUI representations, as opposed to the migration only to Java platform supported by URGenT. In T3, Mathaino analyzes the instances of each user task comparatively to construct an abstract model of:

- The navigational sequence through the legacy system UI to perform the user task;
- The types of information input by the user to the legacy UI and displayed to him/her through his/her navigation, and the locations where they occur on the legacy screens;
- The domain of values of the inputs; and
- The interdependencies among these values.

Note that to analyze the instances of each user task, evidences of the user inputs and outputs is necessary. All user inputs are already recorded in the interaction traces. The CelLEST process needs a user or an analyst to highlight on the snapshots of the task instances the areas that contain the outputs extracted to successfully complete the task.

Given the annotated task instances, Mathaino analyzes the flow of information to and from the legacy system to identify the user inputs required to accomplish the task, by studying all the recorded instances of this task. It compares the values used for each input field across all the task instances, and the values of all input and output fields in the same task instance. Each data input field is labeled with one of the following:

- **constant**, whose value is the same in all task instances,
- **derived**, whose value is obtained earlier in the task from an output field,
- **redundant**, whose value is input multiple times in the same task,
- **enumerated**, whose value is always one of a limited set of values, and
- **unpredictable**, whose value is independently supplied by the user.

Categorizing input fields leads to a significant reduction in the user input required by the reengineered UI of the task, e.g., the user will not need to input a derived input as it will be supplied automatically. Categorization helps choosing the proper abstract widget type for each input field in task T4, e.g., an input labeled with "range" can be implemented using a combo box or a set of radio buttons. CelLEST engineer may inspect the identified pieces of information and name them with meaningful names

Additionally, by comparative analysis of all instances of the same data field, Mathaino infers the coordinates of this field on the legacy screen it belongs to, in case these coordinates are static, i.e., the data filed always appears in the same location of the screen. In dynamic screens, such as free forms, attempts are made to discover starting and/or terminating landmarks to use for locating the data field.

Finally, if alternative paths exist for a user task or subtask, the branching screens need to be manually identified. Then, each alternative path is analyzed as described above. At runtime, the signature (predicate) of the snapshot received after performing an action on an instance of the branching screen decides which path to follow.

The task model produced in T3 specifies the path on the interface state-transition model through which the user navigates, the flow of information between the legacy application and the user, and the syntax of the interactions through which the information is exchanged. Effectively, it constitutes a declarative and executable specification of the modeled task of the legacy application. Given values for all the "unpredictable" pieces of information identified, the model can be used to drive the legacy application and execute the modeled task.

3.4 T4: Generating Abstract GUI Specifications

Mathaino uses model-based UI design heuristics to produce automatically abstract specifications for the new reengineered UI of each task, using its model. Thus, it eliminates the need for the current manual practice of piece-by-piece mapping of the task

model to a GUI design. The specifications are described in terms of a set of abstract forms, each corresponding to a set of screens of the legacy system. Mathaino ensures that all the output fields identified in the task model are displayed on one of the forms. For each abstract form, a corresponding plan for navigating through the legacy screens at runtime is produced.

Using various heuristics, an abstract widget is proposed for each input or output field. For example, a field with an enumerated range of values is represented by a combo box or a set of radio buttons depending on the number of values and an "unpredictable" variable is represented by a text box. Then, the widgets are laid out on the form in a tabular manner. The user can override the default choices of widgets and layout settings. For example, s/he can change the widget type issued for a field or the number of layout columns. After applying user feedback, an XML representation of the abstract specifications is produced.

3.5 T5: Runtime GUI Generation

CelLEST runtime environment consists of two components. The front-end one is the runtime interpreter. It is responsible for interpreting the XML abstract GUI forms on a specific platform. It supplies widgets in the target platform that most closely match the abstract input widgets. Currently, an XHTML interpreter (for Web-enabling) and a WML interpreter (for WAP-enabling) are available [KS01].

The back-end component is the host navigator. It is built over an open source host-access middleware [JM00]. The host navigator executes the navigation plans of the abstract forms and passes the inputs to the legacy system and collects back the outputs. But first, the XHTML or WML interpreter passes the plan details to the host navigator in a platform-independent format.

3.5.1 T5.1: The XHTML Interpreter

For Web-enabling, the XHTML interpreter dynamically parses the XML abstract GUI forms at runtime and translates them to XHTML CGI forms. It maps the abstract GUI widgets to the appropriate CGI widgets. It uses XHTML tables to layout the produced web page in the closest format to that chosen by the user for the abstract GUI. Also, it parses the CGI response produced by the client Web browser into the platform-

independent form needed by the back-end host navigator. It is a server-side component that runs as a Java servlet on the web server.

3.5.2 T5.2: The WML Interpreter

WML was developed by the WAP forum [WAP] for rendering web pages on WAP-enabled mobile Internet devices like Cellular phones and Personal Digital Assistants. A web page in WML (also called WML deck) is limited to a maximum of 1200 bytes. To overcome the device display limitation, a deck can be divided into a number of cards. The device can display only one card at a time. The only input widgets supported by WMP are simple text boxes. WML does not support CGI but provides some features that can simulate CGI.

The WML interpreter is adjusted to deal with these constraints. For example, it implements an abstract GUI form using several WML decks if 1200 bytes are not enough to implement the form. It uses a numerical menu to represent “enumerated” input fields. It internally caches the user responses to the multiple decks corresponding to a single abstract GUI form, before submitting it to the host navigator.