Migration of non-decomposable software systems to the Web using screen proxies

Thierry Bodhuin°*, Enrico Guardabascio* and Maria Tortorella°*

°RCOST - Research Centre on Software Technology
Department of Engineering, University of Sannio
Palazzo Ex-Poste, Via Traiano– 82100, Benevento, Italy
*PST – Parco Scientifico e Tecnologico di Salerno e delle Arie Interne della Campania
Via Portacatena 52, – 84121 Salerno, Italy

(bodhuin@unisannio.it, eguardabascio@pstsa.it, tortorella@unisannio.it)

Abstract

The adoption of the Internet technologies favors the diffusion of Web-based applications. However, the development of new applications exploiting the modern interfaces and distribution channels have to face the problems connected with the integration of existing software systems. This aspect often requires the migration of legacy systems toward the Web technologies. They encapsulate a great deal of knowledge and expertise about the application domain and cannot be simply discarded.

Many of the migration strategies proposed in the literature assume that the underlying software system is (semi)decomposable where the interface component is separated from the business logic and the data model components. In this case, the interface component can be re-implemented, while the other ones are encapsulated into software wrappers. Unfortunately, legacy systems are very often non-decomposable and alternative solutions are needed. This paper proposes an approach for migrating a non-decomposable software system. The approach consists of redirecting the request of input/output operations to the legacy system by using screen and database proxies. The approach has been applied successfully to Cobol software systems with a character-based user interface.

1 Introduction

The Internet and Web technologies adoption can be considered a big evolution factor for the future of legacy systems. However, the evolution had to face the problems connected with the integration of the existing software systems and the new user interface paradigm. Legacy systems cannot be simply discarded as they encapsulate a great quantity of knowledge and expertise about the application domain, which is of great importance for the business. Moreover, the high costs and risks of adopting a new system motivate the choice for an incremental migration strategy [8] from procedural/text-based environment to object-oriented/web-based technologies. This strategy can be based on reverse engineering, wrapping and GUI re-implementation and allows the gradual introduction of object-oriented components in the Enterprise Information System and, on the other hand, extends the benefits of past investments in information technologies [7].

The authors of this paper proposed an incremental migration strategy in [2]. In this strategy, the wrapping allows new object oriented applications to coexist with legacy components. A wrapper represents the interface making new applications to access legacy components through exchange of messages. A wrapper converts the messages into calls to the system components performing the required service. The strategy has been completed with the implementation of a toolkit [6] supporting the migration of a legacy system toward the Web. The applied approach was based on the identification of an object model of the application and its decomposition on the basis of the Model-View-Controller design pattern, and was focused on the persistent data and the separation of the user interface components. Techniques of automatic migration of the user components towards JSP (i.e.: Java Server Pages) and wrapping of the individuated objects were applied.

The cited approach cannot be applied when the system is not clearly decomposable. On the basis of the previous experiences and analysis of other solutions proposed in the literature, the authors have defined the approach presented in this paper. It supports the migration of non-decomposable legacy software systems towards the Web. This kind of systems cannot be easily decomposed into independent components on the basis of the Model-View-Controller design pattern.

The approach proposed in this paper leaves unchanged the control flow of the legacy system and redirects the requests of input/output operations to it by using proxies processing them. This allows using the legacy system in a Web-based environment and the management of its data by using a modern relational DBMS and satisfying the non-functional requirement of reducing the migration costs. The approach requires an automatic transformation of the legacy source code, and a tool supporting the approach has been implemented for Cobol software systems.

The presented research were realized within the research project M&S SW (Methods and Tools for the Production of the Software, Formation and Applications), financed by the Italian Minister of the Scientific Research and carried out by the PST (Scientific and Technological Park) of Salerno, Italy. The project aimed at defining new technological
solutions to be transferred to Small and Medium Enterprises operating in the Information and Communication Technologies.

Related works are presented in the following section; the proposed approach for the migration of legacy systems and supporting tool follows. The fourth section discusses the approach with reference to the COBOL environments. The fifth section presents the results obtained from three case studies, while the last section gives some conclusive remarks.

2 Related works

Sneed [19] proposes a software component encapsulation at different granularity levels (batch process, online transaction, program and module). Each component can be accessed from other foreign environments, through wrappers and by message passing. The same author in [20] describes the tools and technique for encapsulating host COBOL programs behind an XML interface. In [22], Thiran and Hainaut present a wrapping methodology for legacy data system reuse supported by an operational CASE-tool.

In [5], the authors present a pilot project for the migration of a legacy COBOL system to a Web-based architecture, using user interface reengineering and wrapping techniques. The user interface components are isolated and re-implemented using Microsoft Active Server Pages and VBScript script language. The legacy program wrappers are implemented as dynamic loaded libraries written in MicroFocus Object COBOL. In [9], an integrated tool environment, based on static analysis of character-based panels for the identification of dialog entities, is presented. It supports the migration of character-based panels of a mainframe application to graphical user interfaces in a client/server application. In [3], Antoniol et al. propose an approach to reengineer BASIC PC legacy code into modern graphical systems in C++ code. The approach was based on the conceptual representation of the original code in terms of abstract graphical objects and callbacks.

Many of the proposed strategies are referred to decomposable software systems with the interface components separated from the business logic and the data model components. In these cases, the interface component can be re-implemented, while the other ones are encapsulated into a software wrapper.

For semi-decomposable systems in which only the user interface may be separated from the rest of the system, the objects may be wrapped and the user interface may be migrated towards JSP pages (e.g.: keeping the data model and data store unchanged). In [5] a practical migration example of semi-decomposable Microfocus Cobol legacy system is presented.

Differently from the solutions above, the approach proposed in this paper supports the migration of non-decomposable legacy systems, and requires the migration of old character-based user interface by using Web-oriented technologies, a form of Web-based screen scraping [16]. Screen scraping is a common technique for user interface modernization that consists of wrapping old, text-based interfaces with new graphical interfaces. The old interface is often a set of text screens running in a terminal. In contrast, the new interface can be a PC-based, graphical user interface, or even HTML. Screen Scraping techniques [10] may be used when the system is not clearly decomposable. These techniques [4] use the data stream between the terminal user interface and the application as input, in order to recover and transform the user interface information towards a new user interface or in HTML format using a user interface emulator. Several commercial tools [1, 13, 23], using these techniques, allowing the integration of legacy application in Web or windows-based environments without modifying the source code, already exist. The problem of these tools is that they assume that the legacy system is written in a certain programming language and can be executed just within a specific software environment.

In [17], Merlo et al. defined a process to reverse-engineering user interface to obtain its structural and behavioral specification. A prototype implemented the approach and was used to convert and reengineer the user interface for a COBOL/CICS application program.

In [18], the authors presented a restructuring process for user interface components, for migrating legacy applications towards the Web. They use the MORPH technology during the understanding and analysis phases. The proposed process is based on the identification of the user interface components and their transformation towards HTML forms. It keeps on the server side the remaining part of the system and interacts with it by using the Common Gateway Interface (i.e.: CGI). The user interface functionalities are decomposed in two parts: one is used for creating the HTML forms usable for the presentation; and the second is used for processing the input field in the HTML forms and communicating with the legacy system components that uses the CGI interface.

Csaba presents in [11] an approach for user interface reengineering, in order to transform the DOS screens in Windows user interfaces. The approach uses specific Microfocus COBOL applications utilizing ADIS libraries for the management of the user interface input/output operations. It is based on the transformation of the user interface instructions into calls to functions of a C library for windows user interface.

The approach described in this paper is similar to the screen scraping technique [16, 10] and has its same advantages (e.g.: cost, time to market, ...) and disadvantages (flexibility, limited impact on maintainability, ...). Anyway, it is not oriented to a specific software environment like in [1, 13, 23], but the used technologies permit to obtain a portable migrated system. The proposed solution does not rely on the reverse engineering of the user interface as in [12, 9] and does not need a significant work for each new legacy system application. It does not use wrapping techniques for XML-enabling functions of the legacy systems like in [21, 20], as the goal was not to use the legacy system from different contexts but simply to let it be accessible from a Web browser instead of an old character-based terminal. The structure of the legacy system is not modified as in [18] and does not allow the migration to event-driven legacy application but keeps the changes and costs to a strict
minimum. The approach does not require any manual intervention and is completely automated.

3 The migration approach

As already stated, the proposed approach consists of redirecting the requests of input/output operations to the legacy system by using proxies processing them. In particular, the requests can come from the screen or regard the interaction with the file system. A Screen Proxy is introduced for the management of the requests coming from or going to the user interface, while a Database Proxy is used for managing the operations accessing the persistent data.

Figure 1 depicts the new architecture of a legacy system after its migration and the introduction of the Screen and Database Proxies. The user interface migration towards the Web, and the persistent data one towards a relational database are independent and can be conducted separately.

The plain lines in the Figure 1 represent the data exchanges, while the dashed line represents the legacy system activation from the Web. Figure 1 shows that on the user interface side, besides the Screen Proxy, another component is added between the legacy system and the Web Server: the Screen Model. It represents the current state of a character-based screen in terms of a matrix of characters. The Screen Model keeps information about: text content and related format (color and style); input field with its position and size; current cursor position; and special keys state [F1, ..., F12, ctrl, shift].

The numbers on the arrows in Figure 1 indicate the order in which the various components interact. First of all, the user by using a Web browser requests to the Web server services of the legacy system (1). The Web Server activates the first time the legacy system and remains in a blocking state waiting that it sends a response to the user through the Web Browser (2). The input/output requests from the screen (e.g.: ACCEPT/DISPLAY in Cobol) are redirected to the Screen Proxy (5), which allows the communication between the legacy system and the Screen Model (4). The Screen Model keeps in its state a representation of the screen and provides the needed functionalities for communicating with the Web Server (3). In particular, it generates a representation of the screen in HTML format and the synchronization of the interaction between the Web Server and the legacy application, mainly for recuperating the HTML form input fields coming from a Web request and sending them to the legacy system. A HTML page generated by the Screen Model corresponds to a screen of the legacy system user interface, and contains just one HTML form.

On the other side, the instructions of the legacy system for accessing the persistent data are transformed into calls to the Database Proxy (7). Interacting with a Database wrapper, it allows the operations to access the relational database.

The two sections of Figure 1 contained in the dashed rectangles represent two alternative solutions for accessing the legacy data. In fact, the legacy system can continue to access the legacy persistent data store (6) until the migration of the legacy data toward a relational database is completed and the database proxy and wrapper are implemented. At this point, the second solution can be adopted. The two solutions can also coexist if two reciprocally independent sets of legacy data are identified and only one is migrated toward a relational database.

The Web Browser communicates with the Web Server by using the HTTP protocol, and, then, with the legacy system through a java Servlet.

During the execution flow of the legacy system, in correspondence to an output instruction (i.e.: the Cobol instruction DISPLAY) to the screen, a call to the Screen Proxy is performed for executing the insertion of the data to be visualized in the Screen Model. In correspondence to an input instruction (i.e.: the Cobol instruction ACCEPT), a call to the Screen Proxy is performed for permitting to unblock the Web Server and block the execution of the legacy application until a notification arrives indicating that the input field of the Screen Model has been filled with a value coming from the Web browser. The block of the legacy system execution is possible if it happens in the same process of the Screen Proxy and can be synchronized with it. This can be achieved if the Screen Proxy and the migrated legacy system use programming languages that can be integrated.

The interaction happens in the following manner:

- the user fills through the Web browser the HTML form and sends it to the Web server;
- a Java Servlet captures the input field value and transfers it to the Screen Model associated to the user session;
- the Screen Model transfers the value to the legacy application, unblocks the legacy application and waits the notification from it (i.e.: ACCEPT command).

With reference to the migration of data persistent store, the open/close instructions are replaced by connect/disconnect instructions on the relational database, and read/write instructions are replaced by a call to the Database Proxy that operates on the database tables through the Database Wrapper.

The main problem of this approach resides in the communication between the legacy system and the proxies, and the complexity of this problem depends on the programming language used to write the legacy application, and the one used for implementing both the proxies and the other components presented in Figure 1.

It is preferable that they are implemented in an object oriented programming language owning all the utilities for interacting with the Web and accessing the relational databases. The Java language is a first choice for satisfying
these requirements. The legacy system should also be able to perform calls to external programs written in a different programming language, and permitting the parameters exchanges. In this way, the legacy system favors the communication in both ways with the two proxies at user interface and database level.

In this case, the proxy is a class offering the functionalities for exchanging data between the legacy system and the new system, and for interpreting them in order to be able to activate other classes (e.g.: Screen Model, DB Wrapper).

Once defined the communication protocol between the proxy and the legacy system, it is possible to automate the conversion process of the input/output instructions by implementing a tool that individuates the user interface/database input/output instructions and transforms them into calls to the Screen/Database proxies with the opportune parameters.

Besides the proxies establishing the communication between the legacy system and the web relational database, a tool has been implemented for automating the transformation of the source code of the legacy system. This tool has been analyzed and tested for Cobol systems.

4 Applying the approach to Cobol Systems

The strategy described in the previous section has been applied to three legacy systems written in the Cobol programming language. The approach will be detailed only for user interface migration. In the following, the word “Proxy” is used as a synonym of “Screen Proxy”. In this section a general approach valid for each Cobol dialect is presented, while, in the following section the performed case studies regarding real Cobol systems are described.

For the analyzed case studies, the proxies have been implemented by using the Java programming language for gaining the object oriented aspect and Web integration facilities. The communication between the Cobol legacy system and the Java proxies has been reached by using the commercial tool PERCobol [14], a Cobol compiler generating Java classes as target platform. PERCobol includes also all the Java libraries supporting the functionalities of the Cobol language for accessing legacy data repository, like sequential file and indexed file accesses. It supports the calling of both Java code from Cobol and Cobol code from Java with parameters exchanges. Indeed, the Java PERCobol package com.synkronix.cobol contains the Java interface callableProgram that, if implemented by a Java class, provides the possibility to be called by Cobol programs thanks to the method call( parameterList params ). The following subsections presents how the conversion of Cobol is executed and the architecture of the migrated the Cobol legacy system.

4.1 The Cobol code conversion

Figure 2 shows an example of the use of the callableProgram interface. It shows at the top the CalledFromCobol Java code called from the CALLJAVA Cobol program shown at the bottom of the figure. When the control flow reaches the CALL instruction, the control is transferred to the first instruction of the Java method public void call( parameterList params ), and it passes again to the Cobol program after the end of the execution of the Java method. The data exchange between the Cobol program and the Java code is obtained through the USING clause of the CALL instruction in the Cobol code. The exchanged parameters can be either input to or output from the called program and can be passed by reference or by value. The Cobol parameters are transferred as an instance of the parameterList class, which the Java program can access in read and write mode through set and get methods.

```java
import com.synkronix.cobol.callableProgram;
import com.synkronix.cobol.parameterList;
public class CalledFromCobol implements callableProgram {
    public void call() {
    }
    public void call(parameterList params) {
    ...
    }
    public void cancel() {
    }
    public String redirectCall() {
        return null;
    }
}
```

 IDENTIFICATION DIVISION.
 PROGRAM-ID. CALLJAVA.
 ENVIRONMENT DIVISION.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01 PARAM-1 PIC X(6).
 01 PARAM-2.
 02 SUB-1 PIC 99.
 02 SUB-2 PIC 999.
 PROCEDURE DIVISION.
 MAIN-PARAGRAPH.
 ...
 CALL "CalledFromCobol" USING PARAM-1 PARAM-2.
 ...
 END-PARAGRAPH.
 STOP RUN.

Using this capability, it is possible to create a Java class that implements the Proxy, and automatically convert the Cobol ACCEPT and DISPLAY instructions into calls to it. Figure 3 shows the tool structure implementing the automatic conversion.

![Figure 3: Tool structure for Cobol code conversion](image)
The Code Converter tool in Figure 3 takes a Cobol source code as input, and produces in output source code in the PERCobol extension, that can be compiled by the PERCobol compiler for obtaining a Java class. The Code Converter analyzes the source code in input, identifies the Cobol ACCEPT and DISPLAY instructions, recovers all the related information (e.g.: display position on the screen, color, input and output parameters, ...) by using a parser, and replace each of them with a CALL instructions to the Proxy with all the appropriate parameters. Table 1 shows an excerpt of the Microsoft COBOL code and its converted code, obtained by using the code converter.

The table shows that both ACCEPT and DISPLAY instructions are converted into call instructions to the same Java program. The call parameters specify the type of the instruction to be executed (i.e.: ACCEPT or DISPLAY) and a set of information indicating the screen position to which the DISPLAY or ACCEPT instruction are referred, the data fields and the other instruction clauses. The proxy will then interpret the call instruction and communicate with the Screen Model.

The parser internal to the Code Converter is tightly related to a specified COBOL dialect. Therefore, it is necessary to be able to use a set of parsers in order to work with different COBOL dialects.

A problem related to the conversion concerns the technique used to specify the positions on the screen (in terms of column and line numbers). In fact, the example in Table 1 uses directly their values, but they can also be defined by referencing some fields defined in the DATA DIVISION of the COBOL program. In this case, the fields are passed by references.

Another problem met in the COBOL program is related to the use of function keys (F1, F2, ... F12) as terminator of the input instructions. The user can interact with the program and its control flow by using them, but the possible use is different in each COBOL dialects. For instance, several Cobol dialects like RM/Cobol [16] and Microfocus Cobol [15] use the following kind of construction of the ACCEPT instruction:

ACCEPT identifier FROM ESCAPE KEY

It permits to assign to the identified field the code of the function key that has stopped the last input instruction and to continue the execution with the next instruction.

The Microfocus Cobol dialect uses in the SPECIAL-NAMES paragraph the clause

CRT STATUS IS KEY-STATUS

It permits to define a data structure (KEY-STATUS) in the WORKING-STORAGE section that, thanks to the ADIS module, will be automatically updated with the code of the function key that ends the last input instruction on the screen.

The examples considered above are referred to screen input/output instructions that operated on one single-data field. In other cases the screen definitions are defined in the Cobol SCREEN SECTION, used to define a structured screen using labels and input data fields where the positions of the different components are defined in the DATA DIVISION and not during the execution of each single input/output instruction. In this case, it is necessary to recover at the runtime the screen structure, in terms of positions and other visualization attributes. For this purpose, the PERCobol environment provides a valid support through the parameterList class, representing a list of parameters, which are Cobol variables, each of which presenting a hierarchical reference to CICS and 4GL instructions. For making this possible, the PERCobol classes have to be extended for handling screen IO, the presented solution has to be extended in order to consider and convert CICS or 4GL instructions to handle screen IO, the presented solution has to be extended in order to consider and convert them. Unfortunately, this task can be very complex. The alternative adopted solution does not modify the source code and make the Screen Proxy to handle the different kinds of IO instructions. In this case, the Screen Proxy is more complex. However, PerCobol handles part of its functionalities with reference to CICS and 4GL instructions. For making this possible, the PerCobol classes have to be extended for allowing the interaction with the Screen Model.

4.2 The architecture of the new Cobol application

Based on the previous considerations, and referring to COBOL programs, the general schema in the Figure 1 is developed in the Figure 5, where the new architecture of the legacy system is depicted. In the figure, the elliptic forms represent the Java classes implemented or generated by the PERCobol tool. In particular, the PERCobol compiler takes

| Table 1: Code conversion example produced by the Code Converter tool |
|--------------------|--------------------|
| PROCEDURE DIVISION. | PROCEDURE DIVISION: |
| DISPLAY (1 1) "ERASE. | CALL "callProxy" USING "DISPLAY LINE COL 1 1 ERASE" |
| DISPLAY (2 1) PARAM-1. | CALL "callProxy" USING "DISPLAY LINE COL 2 1" PARAM-1. |
| ACCEPT (3 1) PARAM-2 | CALL "callProxy" USING "ACCEPT LINE COL 3 1 AUTO-SKIP UPDATE" PARAM-2. |
| WITH AUTO-SKIP UPDATE. |                        |
|                      |                        |

It permits to assign to the identified field the code of the function key that has stopped the last input instruction and to continue the execution with the next instruction.
as input the Cobol System modified by the Code Converter and generates the classes that belong to the PERCobol System macro-block, as Java byte-codes.

The Screen Model class represents the model of a character-based screen and contains the methods for the communication between the Web browser and the Screen Proxy. The Screen Proxy class allows the communication between the Cobol system and the Screen Model class.

The communication between the Web browser and the Screen Model has been implemented by using a Java Servlet called Screen Servlet. As the Figure 5 shows, the Cobol system is executed with the Screen Proxy in a separate Java Thread, different from the Servlet Thread, for permitting that several users use the Cobol application in the same time, and allowing the synchronization, based on the Screen Model, of the Screen Servlet and the Cobol system. This can be achieved by blocking the Screen Servlet until the Cobol system receives an input instruction, and, then, blocking the Cobol system until a new Web request with the Web session is received.

Each user of the legacy system, sends an HTTP request to the Web Server that manages, by using the Screen Servlet, a Web session associated with the user. A Screen Model instance is associated to each session and a Java Thread that executes a path of execution to the legacy system compiled with PERCobol.

After the connection of a user, the Servlet creates an association between a user session and an execution thread of the legacy system, starting from the main method. Then the Servlet blocks itself and waits for a notification from the Screen Model. When, during the execution of the legacy system, a DISPLAY Proxy call instruction is met, the content of the Screen Model is updated. While, if an ACCEPT Proxy call instruction is met, the Screen Model blocks the Cobol Thread and unblocks the Servlet Thread; then, the Screen Servlet sends back to the Web Browser the HTML page, and waits until a notification is received. The sent HTML page contains a form text input field and several other HTML components simulating the function keys and/or other non-textual input, i.e. F1 key, ESC, etc. The user inserts the textual input in the HTML form and sends it to the Web Server. Considering closed the input operation, The Web Server transfers the data to the Screen Model, that notifies to the Cobol system thread the end of the ACCEPT instruction and blocks the Screen Servlet.

In the Figure 5, printServlet and printerFile are two elements that are not connected to the input and/or output operations for the screen but to the management of the print operations. In fact, the data flow of the legacy system towards the printer is redirected to a temporary file and, when the flow is closed, a call is sent to the printServlet Servlet, which emits the content of the print file in HTML format.
5 Case studies

The approach and the implemented tool have been tested on three Cobol systems used in real contexts. The legacy systems are named Polizze, Terminal and HARRIS, and implemented by using three different Cobol dialects. The first two systems have been provided by the Italian company Serinf S.r.l. for the management of the processes of the Italian enterprise Amoruso S.p.A., operating in the port of Salerno, and managing the containers transport by ship, the third system is used for managing the room booking in an hotel. Accessing the obtained migrated systems through the Web could effectively use them, and the user perceived that the performance is equivalent to the previous legacy system, and each response can be obtained in less than one second.

In the following sections, the results obtained for each case study are provided and, in some cases, examples of the migrated user interfaces are shown.

5.1 The Polizze system

The Polizze system is designed for the management of the insurance contracts for the ship containers for the transport of goods on boats. It is implemented in the Microsoft Cobol dialect and composed of 66 Cobol programs and 116 Cobol COPY files, for a total number of more than 30 thousands lines of code.

The analysis of the Cobol system architecture demonstrated that the Polizze application is not decomposable in terms of the Model, View, Controller, components and not structured, as it contains in the code several GO TO and ALTER statements.

Figure 6 shows the original screen dump of the character-based Cobol application, while Figure 7 contains the Polizze application migrated towards the Web by using the proposed approach.

The usage of the web application is equal to the original one, except for the special keys (e.g.: function keys, ESC, …) that are present in the legacy interface and not available natively within the Web Browser interaction. In fact, a set of links appears on the top and bottom of the HTML page, permitting the interaction with the application by using the mouse. A click on a link labelled F1 is equivalent to the function keystroke F1 in the legacy Cobol application.

Another difference between the two screen dumps regards the current date visualized in the top right part of the screen. This is because that the original application was not year 2000 compatible and the data on the legacy screen shot is blocked to the year ‘99. The text input field in the Web screen has been JavaScript-enhanced for automatically sending the HTML form to the Web Server when the field is fully filled and AUTO-SKIP is used.

5.2 The Terminal system

The Terminal system is used for managing the ship containers on the port terminal of Salerno in Italy. It is implemented in the RM/Cobol 85 dialect, and is composed of 15 Cobol programs and 23 Cobol COPY files, for a total number of more than 10 thousands lines of code.

Even in this case, the preliminary analysis of the system architecture revealed that the application was not decomposable.

The migration process introduced some differences in the management of some instructions. For example, the input termination of the RM/Cobol 85 dialect uses the clause for the ACCEPT instruction

ON EXCEPTION identifier NEXT SENTENCE

The clause permits to assign to an identified field the code of a function key stopping the last input instruction, and permitting the continuation of the program control flow with the next following instruction. For managing the cited clause, it was replaced by the parameter “identifier” to the end of the CALL “callProxy” instruction. The obtained call statement is the following

CALL “callProxy” USING “ACCEPT LINE_COL 2 1” PARAM-1 identifier.

Another characterist of the RM/Cobol dialect is the use of “Pop-up” windows managed by using the clause CONTROL of the DISPLAY instruction. Specifying the position, size, border, and title through the parameters of the DISPLAY instruction, this clause permits the creation/destruction of a screen window. This characteristic has been managed by adding a pool of windows to the Screen Model, having its same features with some parameter more characteristics of the R/M Cobol windows. Exploiting this characteristic, each DISPLAY instruction with the CONTROL clause has been transformed in a special call to the proxy with the exchange of parameters related to the structure of the window. In correspondence of this instruction, the Proxy constructs a suitable window on the Screen Model that is considered the current output for the next instructions.

Figure 8 shows the screen dumps of the web migrated Cobol application of the “Terminal” Cobol system. Even in this case, the screen dumps of the Web application are identical to the original ones. The figure exhibits the “pop-up” windows of RM/Cobol dialect

5.3 The HARRIS system

The system named HARRIS, Halls, Residences Record and Information System for the management of residences for students, was developed in the Microfocus Cobol dialect. It was composed of 101 Cobol programs and 88 Cobol COPY files, for a total number of more than 38 thousands lines of code.

The analysis of the system architecture highlighted that the HARRIS application was a decomposable application thanks to the good separation between the user interface, data accesses and control functionalities, and the extensive use of the Cobol SCREEN SECTIONs. This system has also been used for testing the migration strategy defined in [6]. For managing the input termination, the Microfocus Cobol dialect uses “CRT STATUS IS KEY-STATUS” in the
SPECIAL-NAMES paragraph, permitting the definition of a data structure, KEY-STATUS, in the WORKING-STORAGE section. This will be automatically updated, thanks to the ADIS module, with the function key code ending the last input instruction on the screen.

Figure 6: User interface from the “Polizze” system

Figure 7: New Web user interface for the “Polizze” system
The clause “CRT STATUS” for Microfocus Cobol is managed by using the call statement “CALL “callProxy” USING “ACCEPT KEY” identifier” after every instruction of input type. In this manner, the Cobol system requests the code of the terminator keys ending the current ACCEPT statement to the Screen Model.

Figure 9 shows the web migrated HARRIS application with a screen relative to a data structure defined in a SCREEN SECTION with multi-inputs where one input field is presented each time and in sequence.

It can be noticed in the bottom the ALT, CTRL and SHIFT links usable for the communication with the legacy Cobol system that are usable in combination with other keys.

Similar results are obtained for the 75 other screens of the application.

6 Concluding remarks

In this paper an approach for migration legacy systems toward the Web and relational databases, and the related toolkit are presented.

The aim of the migration approach is to provide a tool for automating the source code migration of non-decomposable legacy systems, and the Java classes for its Web integration. The approach does not implement substantial change to the
user interface, so that the user can recognize the old system and continue to interact with it without requiring adaptation interventions. Keeping the old interface permits to satisfy the non-functional requirement of reducing the migration costs. Nevertheless, it does not represent a necessary requirement of the migration approach, and a new intervention may be needed for making the user interface more attractive and dynamic. In the meantime, it is considered as an opportunity helping to win the user resistance to the innovations.

A tool and supporting Java classes implementing the chosen migration approach were illustrated. The approach proposed and the tools were validated in three medium-size case studies whose results were presented. In particular, two of the analyzed software systems have been provided by an industrial partner, and are used in their daily activities for the management of the sailing boats, in the port of Salerno in Italy.

The analyzed legacy systems are written in the COBOL programming language. The encountered problems are connected to the many existing different Cobol dialects that ask for the implementation of a parser, different for each recognized dialect. The principal problem was met for implementing the tool that automatically convert all the possible DISPLAY and ACCEPT syntax and semantic as it requires the grammar and semantic of all the Cobol language extensions that are used.

The next work, that may involve the authors, regards the completion of the definition of the approach, with reference to other programming languages too, and the full implementation of the data migration tool. The final work will regard the continuation of the experimentation of the approach and supporting tool.

Acknowledgement

The authors would like to thank the anonymous referees for the precious suggestions for improving the paper.

References