Available online at www.sciencedirect.com

ScienceDirect

The Journal of Systems and Software 81 (2008) 463–480

The Journal of Systems and Software

www.elsevier.com/locate/jss

A wrapping approach for migrating legacy system interactive functionalities to Service Oriented Architectures

Gerardo Canfora [a], Anna Rita Fasolino [b], Gianni Frattolillo [b], Porfirio Tramontana [b,*]

[a] *RCOST – Research Centre on Software Technology, University of Sannio, Palazzo ex Poste, via Traiano, 82100 Benevento, Italy*
[b] *Dipartimento di Informatica e Sistemistica, Università di Napoli Federico II, Via Claudio, 21, 80125 Napoli, Italy*

Available online 28 June 2007

Abstract

Software systems modernisation using Service Oriented Architectures (SOAs) and Web Services represents a valuable option for extending the lifetime of mission-critical legacy systems.

This paper presents a black-box modernisation approach for exposing interactive functionalities of legacy systems as Services. The problem of transforming the original user interface of the system into the request/response interface of a SOA is solved by a wrapper that is able to interact with the system on behalf of the user. The wrapper behaviour is defined in the form of Finite State Machines retrievable by black-box reverse engineering of the human–computer interface.

The paper describes our wrapper-based migration process and discusses the results of case studies showing process effectiveness and quality of resulting services.

Keywords: Legacy system migration; Service Oriented Architecture; Web Service; Legacy system wrapping

1. Introduction

According to the OASIS reference model (OASIS, 2006), Service Oriented Architecture (SOA) is a paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains. The basic idea of SOA is that a system is designed and implemented using loosely coupled software services with defined interfaces that can be accessed without any knowledge of their implementation platform. Web Services (World Wide Web Consortium, 2004) represent the most common form of SOA where service interfaces are described using the Web Service Definition Language (WSDL) (World Wide Web Consortium, 2001), the Simple Object Access Protocol (i.e., SOAP) is used to transmit data over the HTTP (World Wide Web Consortium, 2003), and the Universal Description, Discovery and Integration (i.e., UDDI) is optionally used as the directory service (OASIS, 2004).

By overcoming interoperability limitations, SOA allows existing software systems to be integrated by exploiting the pervasive infrastructure of the World Wide Web and offers a new chance to continue to use and reuse the business functions provided by legacy systems. Indeed, several organizations are considering the opportunity of modernising their software systems by presenting them as services for exploiting the many advantages offered by SOAs.

However, the migration of a legacy system towards a Service Oriented Architecture is not a straightforward task. It entails the solution of both a decision problem for establishing *what* can be migrated from the original legacy system, and of a technical problem concerning *how* the migration can be executed. As to the first problem, structured approaches that can support organizations analysis of their systems to find candidates for a Web Service are

* Corresponding author. Tel.: +39 081 7683901; fax: +39 081 7683816.
E-mail addresses: canfora@unisannio.it (G. Canfora), fasolino@unina.it (A.R. Fasolino), gianni.frattolillo@gmail.com (G. Frattolillo), ptramont@unina.it (P. Tramontana).

needed: Harry Sneed describes a number of factors to be taken into account for the identification of candidate Web Services in existing systems (Sneed, 2006), while Lewis, Morris and Smith from the SEI propose the SMART (Service-Oriented Migration and Reuse Technique) approach for gathering the needed information and identifying the risks of a migration effort in a systematic way (Lewis et al., 2006).

As to the technical problem, methods techniques and tools that allow the migration process to be carried out effectively are needed. Zhang and Yang (2004) have identified three classes of approaches for integrating legacy systems in SOAs: the first class comprises black-box reengineering techniques, which integrate systems via adaptors that wrap legacy code and data and allow the application to be invoked as a service. A second class includes white-box methods, which require code analysis and modification to obtain the code components of the system to be presented as Web Services, while a third class of grey-box techniques combine wrapping and white-box approaches for integrating those parts of the system with a high business value.

White-box approaches are usually more invasive to legacy systems than black-box ones, but they are able to prolong the lifetime of a system, saving on maintenance, and improving the efficiency of processes relying on legacy code. However, white-box approaches may not be suitable for integrating systems whose code is not available, or systems affected by high obsolescence and deterioration, characterised by low component modularity, a high degree of coupling between components, or a high dependence on their execution environment. In these cases, black-box approaches that leave the original system configuration untouched, as well as its execution environment, may represent the most practicable solution for migration.

A black-box technique for migrating a legacy system towards SOAs has recently been proposed by Canfora et al. (2006). This technique aims at applying a new Web Service interface to *interactive* legacy systems, which are systems where any transaction is completed by means of a dialogue between the user and the system, which may involve browsing for information, selecting items, answering queries and so on. The basic aim of this approach is to provide the legacy system with the request/response interface typical of Web Services, where a client party invokes a service implemented by a provider party using a request message and the provider processes the request and sends a response message with the produced results. The solution proposed to this problem employs a wrapper whose responsibility is to interact with the legacy system in order to complete each transaction on behalf of the user. The feasibility of this wrapping technique has been assessed in a preliminary case study of software migration.

In this paper, we provide a more detailed description of the migration approach and discuss the results of an extended experimentation consisting of two different case studies. The paper is organized as follows: Section 2 describes related works on legacy system modernisation and migration to the Web, while Section 3 presents the migration problem addressed in the paper and the proposed wrapping solution. Section 4 shows the characteristics of the software wrapper, while Section 5 describes the steps of a migration process aiming at reusing user functionalities implemented by legacy systems. Section 6 discusses two case studies of migrating a legacy application and, finally, Section 7 provides concluding remarks and outlines directions for future work.

2. Related work

Exposing existing software systems as services in a Service Oriented Architecture can be considered as a possible approach for managing a legacy system. A legacy system can be defined as 'a mission-critical software system developed sometime in the past that has been around and has changed for a long time without undergoing systematic remedial actions' (De Lucia et al., 2001). The real challenge of legacy systems consists of finding cost-effective and quality solutions for evolving them in order to meet new requirements (Bennett, 1995).

Modernising a legacy system (Comella-Dorda et al., 2000) is an applicable option whenever the system requires more pervasive changes than those possible during ordinary maintenance (IEEE, 1998), but still has a business value that must be preserved. According to the classification proposed by Bisbal et al. (1999), three main modernisation techniques are applicable to legacy systems: *Redevelopment*, that requires the system to be redeveloped from scratch, using a new hardware platform and modern architectures tools and database; *Wrapping,* which surrounds existing data, programs, applications and interfaces with new interfaces; and *Migration* that moves the system to a new platform, while retaining the original system data and functionalities.

Comella-Dorda et al. (2000) pointed out the distinction between *white-box modernisation*, based on a knowledge of the internal workings of a legacy system, and *black-box* modernisation, requiring just the knowledge of the external interfaces and often based on wrapping. Moreover, the same authors distinguished between modernisation techniques involving the user interface, the data, or the functionalities of the legacy system.

In the past years, the problem of modernising a legacy system by exporting it to the Web was addressed by several authors in the literature who concentrated their efforts on three main research topics: the reengineering of a traditional (i.e., not Web-based) system into a Web-based one, the migration of Web applications to Web Services, and the migration of traditional applications towards Web Services.

As to the reengineering of a traditional system into a Web-based application, Zdun (2002) identified four main steps in a possible migration process and proposed a reference architecture for bringing a legacy application to the Web. Harry Sneed (2001, 2003) discussed reengineering solutions based on wrapping for encapsulating host COBOL programs with an XML interface, while Bi et al. (2002) explored the possibility of migrating a legacy application to the Internet-based platform using XML and distributed object technology (such as Java RMI and JNI) and a wrapping approach exploiting the knowledge of users' interactions with the original systems. Stroulia et al. (2002) presented a method based on interaction modelling for migrating a legacy system to the Web in the context of the CelLEST project. Moreover, Bovenzi et al. (2003) have described a solution based on wrapping that can make existing interactive systems accessible by heterogeneous client devices using the common infrastructure offered by the Web. This solution, however, did not address the problem of transforming the legacy system interactive interface into a request/response-based one, but just its migration to a different interactive client interface.

A further family of approaches for migrating a traditional system towards the Web is based on screen scraping techniques. Screen scraping (Comella-Dorda et al., 2000) consists of wrapping old, text-based user interfaces with new graphical interfaces, which may be implemented in HTML and accessed by a Web browser. The new graphical interface communicates with the old one using a specialised commercial tool. These tools often generate the new screens automatically by mapping the old ones. Examples of these tools include the *IBM WebFacing Tool* (Nartovich et al., 2004), allowing legacy applications based on 5250 terminals to be accessed via web pages; *IBM WebSphere Host Publisher* (Rodriguez et al., 2001), and the Jagacy (Jagacy Software, 2006) tool that provide access to stream-oriented or block-oriented legacy terminals from Java applications. Finally, Novell exteNd (Novell, 2006) is a tool that allows the abstraction of legacy system terminals as Javascript objects. Of course, in all these cases screen scraping is used for migrating an old interactive interface into a new, Web-based but still interactive interface, since it is not able to transform alone the interactive paradigm of the old interface into a request/response-based one.

Some authors have recently presented their experiences with the migration of Web applications to Web Services.

Guo et al. (2005) propose a white-box reverse engineering technique and a tool for generating wrapper components that make the functionalities of a Client–Server .NET application available as Web Services. Jiang and Stroulia (2004) describe their work constructing Web Services from functionalities offered by Web sites: their approach exploits a reverse engineering technique for modelling the interaction of the user with Web sites and obtaining the functionalities to be specified in terms of WSDL specifications. Baumgartner et al. (2004) have proposed a suite for obtaining Web Services from applications based on Web-based interfaces: the suite includes a visual tool that allows the extraction of relevant information from HTML documents and translation of them into XML which can be used in the context of Web Services.

As to the third topic, i.e., the migration of traditional applications towards Web Services, Tilley et al. (2002) discussed some challenges inherent to the transformation of existing programs into Web Services accessible by a WSDL interface, while Litoiu (2004) addressed the problems of latency and scalability in migrating to Web Services traditional applications using a UML-based solution. Zhang and Yang (2004) have proposed a grey-box reengineering technique based on clustering to extract services from legacy systems, while Sneed (2006) has recently proposed a tool-supported process for creating Web Services from legacy code using a wrapping approach. This approach requires code analysis and manipulation for obtaining the parts of the legacy application to be exposed as Web Services; therefore it must be considered as a white-box reengineering approach.

The modernisation technique proposed in this paper is a migration technique that allows selected user functionalities implemented by an interactive legacy system to be exported as Web Services. Unlike other approaches proposed in the literature, this technique is not invasive for the legacy system but just requires an understanding of the system interfaces so as to produce a wrapper that allows a system functionality to be used as a Web Service.

The problem of understanding the interfaces of an existing software system can be considered as a classical reverse engineering topic, approachable either by white-box or black-box techniques. As to the white-box approaches, some remarkable contributions include the reverse engineering technique proposed by Merlo et al. (1995) that aims at extracting high-level representations of interface elements for reengineering the User Interface of the original system, the solution proposed by Rugaber (1999) for automatically obtaining interfaces specialised for a particular device, as well as the approach presented by Gaeremynck et al. (2003) for abstracting platform-independent models of the User Interface. Black-box techniques, rather than analysing the application code, consider the inputs and outputs during the user interaction with the system to gain an understanding of the system interfaces. An example of these techniques is described by Stroulia et al. (1999) who proposed a process for obtaining a model of the user tasks supported by the system based on screen features analysis and on the tracing of user interactions with the system. A hybrid approach combining black-box and white-box reverse engineering techniques was presented by Chan et al. (2003), in which analysis of the actions dynamically performed on an interactive legacy system and of the returned screens supports the mapping of user actions into the source code of the legacy system. More recently, Draheim et al. (2005) have proposed a black-box reverse engineering approach for dynamic Web Applications, where the screen identification problem has been precisely characterised.

3. The migration problem and the proposed solution

Form-based systems are a class of interactive systems in which human–computer interaction is session-based and composed of an alternating exchange of messages between the user and the computer: the user submits some input on an input screen, the system executes some data processing and responds by presenting a new screen (Atkins et al., 1999), (Draheim and Weber, 2005). Therefore, the user–system dialogue, carried out by a set of forms, is fundamental for completing any transaction.

A large number of legacy systems belong to this class and the migration approach described in this paper is applicable to them. The main difference between a form-based system and a Web Service consists in their external interfaces: while the dialogue is essential for exercising the business rules implemented by these systems, in SOAs a service is obtainable through a simple request/response interface where a Service Request Message (including input data) is sent from a client to a Web Service, and a Service Response Message including output data is sent from the Web Service to the client. Fig. 1 illustrates this mechanism for a simple Web Service.

As a consequence, if a functionality offered by an interactive legacy system must be exposed as a Web Service, the basic issue consists of transforming the original system interface into a request/response-based one. This transformation cannot be addressed through simple screen scraping solutions because these techniques do not support alone the needed interface paradigm change from interactive to request/response-based. The solution proposed in this paper is more complex and is based on a wrapper that encapsulates the original user interface having the responsibility for interacting autonomously with the legacy system and on behalf of the user during the execution of each functionality. This solution is illustrated in Fig. 2.

To carry out this interaction, the wrapper must be aware of the dialogue rules comprised in a transaction. Moreover, since the response of the system generally depends both on user input and on the system's internal state, the wrapper must be able to manage all alternative flows of actions that may occur during the functionality execution. To reach this aim, the wrapper (that just has a black-box view of the legacy system) will have to identify the steps actually reached during the interaction and the corresponding execution scenario by analysing the screens returned by the system.

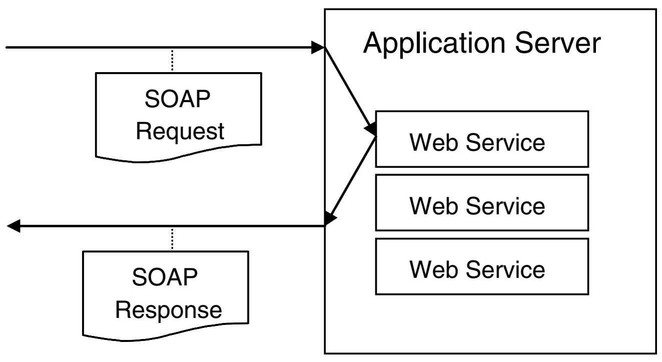


Fig. 1. Web Service Architecture.

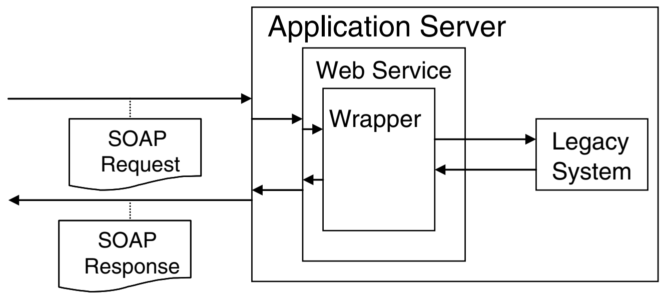


Fig. 2. Wrapper-based architecture for the migration of a legacy system to Web Services.

To satisfy these requirements, a model of the human–computer interaction interpretable by the wrapper, specifying both static (i.e., structure of the UI forms) and dynamic aspects of the legacy system user interface, is needed. Finite State Automata provide a suitable model for specifying user–system interactions and we decided to adopt them in our approach. The details of this FSA-based Legacy System Interaction Model will be described in the next section.

3.1. Modelling the interaction with a form-based system

A finite state machine or finite state automaton is a model of behaviour composed of states, transitions and actions. A state stores information about the past, i.e. it reflects the changes from the system start to the present moment. A transition indicates a state change and is described by a condition that would need to be fulfilled to enable the transition. An action is a description of an activity that has to be performed at a given moment.

State machines have long been used for describing software user interfaces: see Parnas (1969), Wasserman (1985), Olsen (1984) Berstel et al. (2005) for some examples. In form-based systems, they can be used for modelling the human–computer interaction, provided that machine states are associated with the different types of screens observable on the interface, while actions performed by the user on these screens (i.e., messages sent from the user to the system) cause the state transitions. As observed by Draheim and Weber (2005), there is a finite set of screen types observable on this type of interface, there is a finite set of message types between user and system that trigger state transitions, and system's response is conditional, depending both on the message and on the system's internal state.

The Legacy System Interaction Model adopted by our migration approach associates each user-oriented functionality offered by the system (i.e., a use case) with a *Finite State Automaton* FSA $= (S, T, A, S_{\text{in}}, S_{\text{fin}})$, where S is the set of Interaction States reached during the dialogue (associated with a different type of screen), T is the set of transitions between states, and A is the set of types of actions performed by the user on a screen and causing the transition between states. Finally, S_{in} and S_{fin} represent the initial and final states of the interaction. In case of systems whose response to a user message is conditional and depends both on the type of message and on the system's internal state, the FSA will present no transitions, or more than one transition, from a given state for a given input. This property of the FSA is generally called non-determinism, and Non-Deterministic Finite State Automata are the corresponding class of automata (Sipser, 1997). This automaton can always be transformed into an equivalent deterministic one, although this transformation typically significantly increases the complexity of the automaton and requires a knowledge of the system's internal state and of the resources (such as files, databases, etc.) it manages.

FSAs can be represented using state diagrams. Fig. 3 shows an example of a non-deterministic FSA for a 'View Bank Account status' functionality. The FSA includes four interactions states, namely *Authentication*, *Menu*, *Access Denied*, and *View Bank Account*. In the Authentication state, the user can edit the Login and Password fields and press ⟨Enter⟩. Therefore, the system processes the request and, depending on the correctness of the Login and Password input, returns the Main Menu screen or an 'Access Denied' message screen. In the Main Menu the user sends a 'V' character on an edit field and therefore obtains a screen with the Bank account status, while from the Access Denied screen s/he reaches the end screen by introducing a 'Q' character in an appropriate input field.

The Interaction Model needed by the wrapper for its execution will not only represent the dynamics of the interaction, but also the content and the layout of each type of screen observable by the user at each step in the interaction: the screen features that are relevant for an automatic identification of the screen type will be specified by a model called *Screen Template*. Both textual and graphic features may be included in a Screen Template.

Focusing on textual characteristics, a Screen Template can be characterised by a set of text fields, that can be input fields, output fields, or label fields, and by their position on the screen. Input Fields are edit fields where a user can insert an input value, Output Fields are screen areas where an output value is displayed, and Labels are screen areas containing a constant text string. The position of the field on the screen can be defined in two different ways: as a Fixed Location, i.e. by specifying the absolute and constant values of the x and y coordinates where it is displayed on the Legacy Screen; or as a Relative Location, i.e., by specifying the field displacement (or offset) from another field.

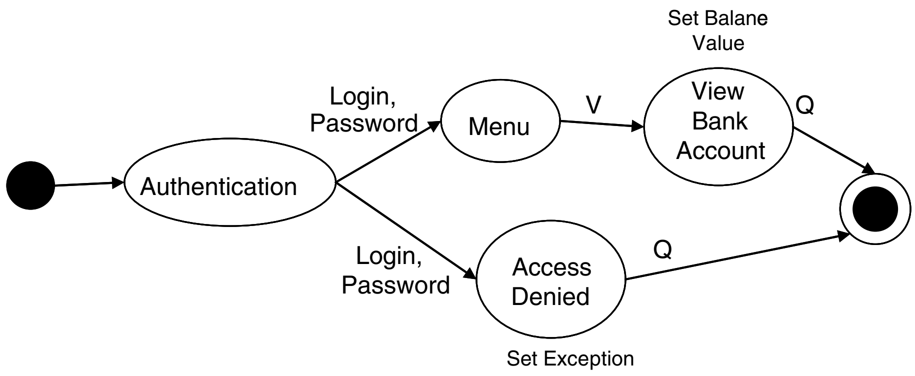


Fig. 3. The Automaton of the 'View Bank Account status' functionality.

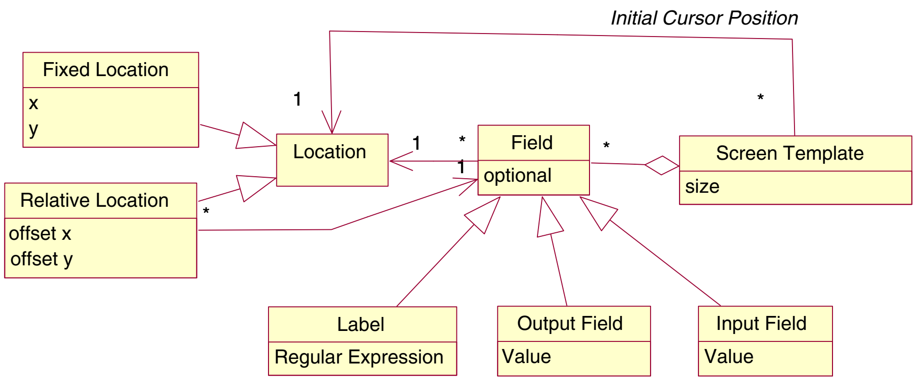


Fig. 4. Screen template model.

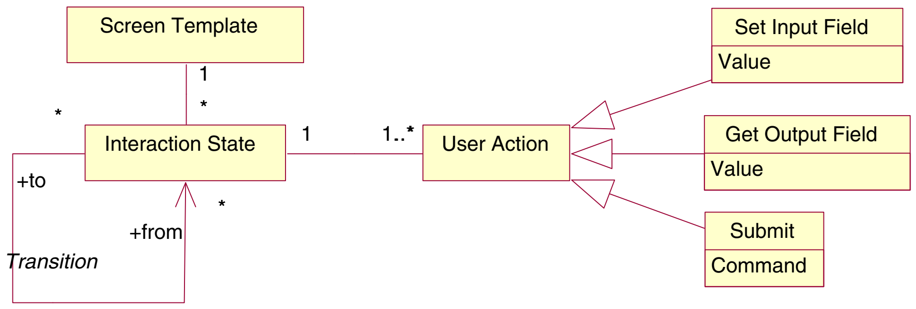


Fig. 5. User interaction model.

This model of Screen Template is described by the class diagram in Fig. 4 that shows the relevant entities of a Screen Template and their relationships.

Fig. 5 presents a class diagram providing a conceptual model of the interaction between the user and the legacy system. This model shows that each *Interaction State* is associated with just one *Screen Template*, but different *Interaction States* may be associated with the same *Screen Template*. Moreover, each *Interaction State* is associated with the set of *User Actions* (such as *Set Input field*, *Get Output Field*, *Submit a Command*) that can be performed on the screen and trigger the transition from that state to another one. Finally, the self-association named *Transition* between *Interaction States* models the transitions between states.

4. The wrapper

In the previous section, the main requirements of the wrapper and the description of the Legacy System Interaction Model have been presented. In this section, the design of the wrapper we implemented, including its software architecture and the specification of its behaviour, will be provided.

4.1. A reference architecture for the wrapper

The wrapper must be able to replace the user during the interaction with the legacy system by exploiting a knowledge of the legacy system human–computer interface. To reach this aim, the wrapper can be implemented as an *automaton engine* that, by interpreting the FSA model provided, will reproduce the user behaviour.

During the interpretation, the automaton engine evolves between the FSA states and in each state it exchanges data and commands with the legacy system and has to buffer the intermediate results of the use case execution that will be needed by the wrapper for composing the final response message. Moreover, when entering a new state, the wrapper will have to determine which use case interaction scenario the current state belongs to and will perform the actions associated with that state.

To satisfy these specific requirements and in order to obtain an adaptable and reusable solution, the software architecture of the wrapper has been decomposed into three main modules, namely the *Automaton Engine*, the *Terminal Emulator*, and the *State Identifier* components. These modules rely on a *Repository*, which persistently stores both the Finite State Automaton and the corresponding Screen Templates interpretable descriptions associated with the use cases. This solution allows the wrapper components to be reusable, since they are decoupled from specific details of the use cases to be migrated which are, vice-versa, embedded in the Repository files.

Besides the Automaton Engine, that is the core of the wrapper and acts as an automaton interpreter, the State Identifier is the module that determines the current interaction state by analysing the last screen returned by the legacy system. The Terminal Emulator module implements the abstraction of the type of Terminal used for interacting with the legacy system, and therefore it manages the user–system flow of data and commands on behalf of the user. The implementation of this last component is dependent on the specific technologies and protocols adopted by different classes of terminals for communicating with a legacy system. Therefore, this is the only architecture component that needs to be changed as a legacy system with a new class of terminals has to be migrated.

Fig. 6 shows the logical architecture of the wrapper and reports the collaborations between these modules and the main flows of data among them. Additional details about each module of the wrapper architecture will be provided in the following subsections.

4.2. Terminal emulator

The Terminal Emulator module is responsible for implementing an abstraction of the type of Terminal used for accessing the legacy system from the externals. Usually the functionality of a legacy system can be accessed either by an API (Application Programming Interface) offered by the system, or a UI (User Interface), or a proprietary protocol. If the interaction with the system is not based on an API but on a User Interface, like most form-based systems, the legacy system uses a *terminal server* to support specific types of terminals using their communication protocols. Two main classes of terminal are generally used: the *block-oriented* class (such as the terminals *tn3270* and

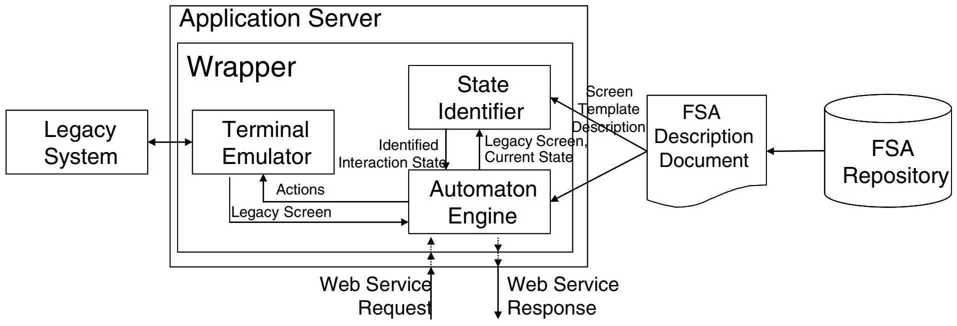


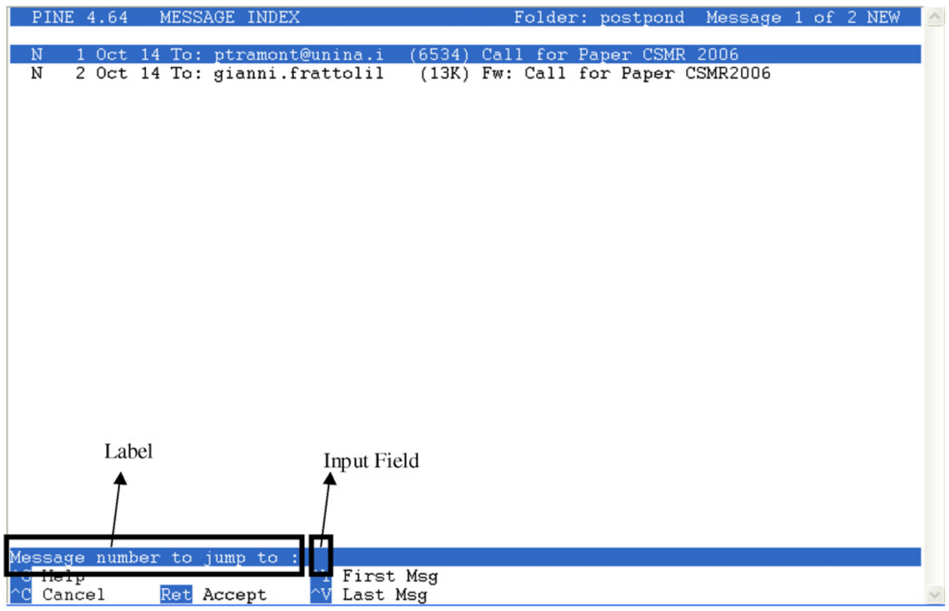
Fig. 6. The architecture of the wrapper.

tn5250, used for the connection with *Ibm Cics* and *As/400* systems) and the *stream-oriented* one (such as the terminals of the *Vt* family: *Vt52*, *Vt100*, *Vt220*, *Vt320*, *Vt420*). These classes are conceptually different both as regards the interaction model and the communication protocol used to exchange data. In particular, *Stream-oriented* terminals may exchange data with the system using a two-directional, byte-oriented communication channel. The input data entered by the operator are sent to the system as soon as they are typed. The legacy system performs the required computation and then sends the output back to the terminal. In these terminals, the screen is organized as a matrix of characters. Vice-versa, *Block-oriented* terminals abstract the legacy screen as a collection of text fields. Each field on the screen has a starting position and individual attributes (such as colours, position on the screen, etc.). The terminal communicates with the system by sending a collection of fields and a command; the legacy system answers this request by another collection of fields.

Whatever the considered class of terminals, the corresponding Terminal Emulator will provide a programming interface for the standard read/write operations executable on the legacy system input/output devices. This interface will be used by the Automaton Engine to interact with the legacy system. The implementation of such an interface will depend on the specific legacy system terminal server protocol and may be partially based on screen scraping techniques that automatically extract data from interactive screens without user interventions.

4.3. State identifier

During its evolution the legacy system evolves between the different interaction states modelled by the FSA. Generally, given a starting interaction state and an input action, several new interaction states can be reached. Each interaction state is, however, associated with one Screen Template. The responsibility of the State Identifier component is to recognize which Screen Template associated to these possible States matches with the Screen returned by the Legacy System. To implement this task, the State Identifier exploits the descriptions of the templates included in the Repository.


```
PINE 4.64   MESSAGE INDEX                    Folder: postpond  Message 1 of 2 NEW

N   1 Oct 14 To: ptramont@unina.i   (6534) Call for Paper CSMR 2006
N   2 Oct 14 To: gianni.frattolil    (13K) Fw: Call for Paper CSMR2006

Message number to jump to :
^G Help                              ^Y First Msg
^C Cancel      Ret Accept            ^V Last Msg
```

Fig. 7. An instance of the “Go To Message” screen template.

As an example, the State Identifier must be able to match the screen reported in Fig. 7 with a Screen Template including the following features: a label, whose value is "*Message number to jump to*" that is positioned at the beginning of row 22 of the screen; and an input field, named *MessageNumber*, that is positioned on row 22 of the screen, from the column 25 to the column 29. The XML description of this Screen Template is reported in Fig. 8. The description also contains the report that a transition is triggered by the submit-command obtained by pressing the carriage return key (i.e., the "*n*" Escape sequence).

Of course, the success matching rate of the State Identifier component will depend on the accuracy of the descriptions of the Screen Templates that can be obtained by reverse engineering the user interface. Defining techniques for automatically or semi-automatically abstracting accurate Screen Templates is an interesting open issue to be investigated.

The State Identifier and the Automaton Engine components exchange messages through synchronous communication: the Automaton Engine sends the current screen description (obtained from the Terminal Emulator) to the State Identifier and waits until the State Identifier responds with the identified interaction state and with the content of the screen, including the values stored in its Label, Input and Output Fields.

```
<screen id="GoToMessage">
  <label id="GoToMessageId">
     <fixed-location>
        <point x="0" y="22"/>
     </fixed-location>
  <content text="Message number to jump to" length="25"/>
  </label>
  <input-field id="MessageNumber">
       <fixed-location>
          <point x="25" y="22"/>
       </fixed-location>
      <content value="" length="5"/>
  </input-field>
  <submit-command>
           <command value="\n">
  </submit-command>
</screen>
```

Fig. 8. An excerpt of the XML automaton description document associated with the "Go To Message" screen template.

4.4. Automaton engine

The core of the wrapper is the Automaton Engine (AE), that coordinates the interaction with the legacy system by interpreting the FSA descriptions and exchanging messages with the other wrapper components.

The execution of the Automaton Engine is launched by the Application Server that hosts the wrapper and provides it with the identifier (i.e., the URI) of the requested service and with the service input data. For each service there will be an FSA description document stored in the Repository that will be read and interpreted by the AE.

The algorithm implemented by the AE is reported in Fig. 9 as a UML activity diagram that includes three fundamental activities, i.e. *Start* activity, *Final* activity, and *Interpretation* activity.

The *Start* activity includes the following actions: the service request message is received through the application server, the corresponding XML automaton description file is read from the Repository, the execution of the legacy system is launched, a set of preliminary initialisation actions (involving the buffer variables called *Automaton Variables*) is performed, the first screen returned by the legacy is received through the Terminal Emulator, and the Current State of the interaction is obtained through the State Identifier component.

Therefore, the engine enters the *Interpretation* activity. In this activity, while the current state is not the Final state of the interaction, the engine executes the actions that are associated to the current interaction state (involving both the automaton variables and the input/output fields of the legacy screen), submits the new input and commands to the legacy system, and waits for the new screen indicating that a new interaction state has been reached. The Current interaction state is, therefore, up-to-date thanks to the State Identifier intervention. When the current state coincides with the Final state of the computation, the wrapper leaves the Interpretation activity and enters the Final one.

In the Final activity, the wrapper composes the Service Response Message on the basis of the data values stored in the automaton variables and sends it to the Application Server.

4.5. Repository

The Repository stores in a persistent manner both the Finite State Automaton interpretable description, and the corresponding Screen Template descriptions for each ser-

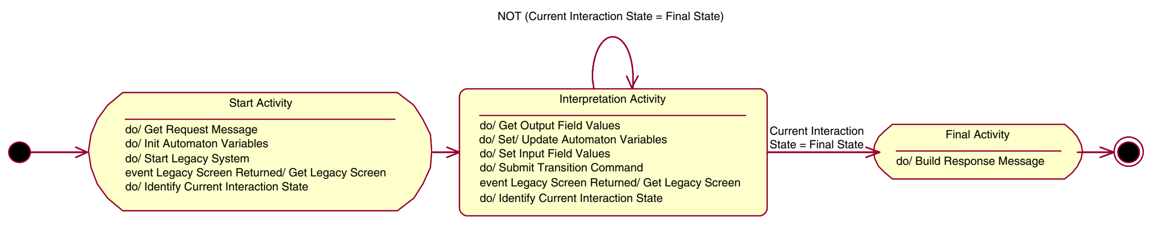

NOT (Current Interaction State = Final State)
Start Activity
do/ Get Request Message
do/ Init Automaton Variables
do/ Start Legacy System
event Legacy Screen Returned/ Get Legacy Screen
do/ Identify Current Interaction State
Interpretation Activity
do/ Get Output Field Values
do/ Set/ Update Automaton Variables
do/ Set Input Field Values
do/ Submit Transition Command
event Legacy Screen Returned/ Get Legacy Screen
do/ Identify Current Interaction State
Current Interaction State = Final State
Final Activity
do/ Build Response Message

Fig. 9. Automaton engine behaviour.

vice offered by the legacy system. These descriptions are stored as XML files, called *FSA Description Documents*. At a conceptual level, the information contained in each file includes the Interaction States, Transitions, Screen Templates, and Actions that will be executed by the wrapper, as well as the list of Automaton Variables that buffer intermediate data and that are needed to implement a given service.

Fig. 10 reports a class diagram describing the model of the information contained in the FSA Description Document. Besides the information about the UI Screen Templates and Interaction states, different types of Actions that can be performed by the Automaton Engine are represented in the model. These Actions include:

- *Init Action*, for setting the Automaton Variables on the basis of the service request message. This action has to be performed in the Start Activity of the Automaton Engine.
- *Set Input Field Action*, setting the value of an Input Field.
- *Get Output Field Action*, reading the values of Output Fields from a legacy system screen.
- *Submit Command Action*, triggering a transition from an Interaction State to another.
- *Set/Update Automaton Variable Actions*, consisting of assignment expressions modifying the actual value of an Automaton Variable, on the basis of Output Fields values and of other Automaton Variables.
- *Build Response Actions*, writing the service response message on the basis of the actual values of the Automaton Variables. This action has to be performed in the Final Activity of the Automaton Engine.

All the actions, with the exception of the Init Action and the Build Response Action, have to be performed during the Interpretation Activity of the Automaton Engine.

5. The migration process

The wrapping technique presented in this paper can be used in the context of a migration process aiming at reusing user functionalities implemented by legacy systems. Such a migration process will include the following phases:

1. Selecting the candidate services.
2. Wrapping the selected use cases.
3. Deploying and validating the wrapped use case.

Each phase will be analysed in the next subsections.

5.1. Selecting candidate services

In the first phase, a decision problem will have to be solved to determine which legacy system use cases can be exposed as services in a SOA. To solve this problem, criteria, methods and structured processes that can support

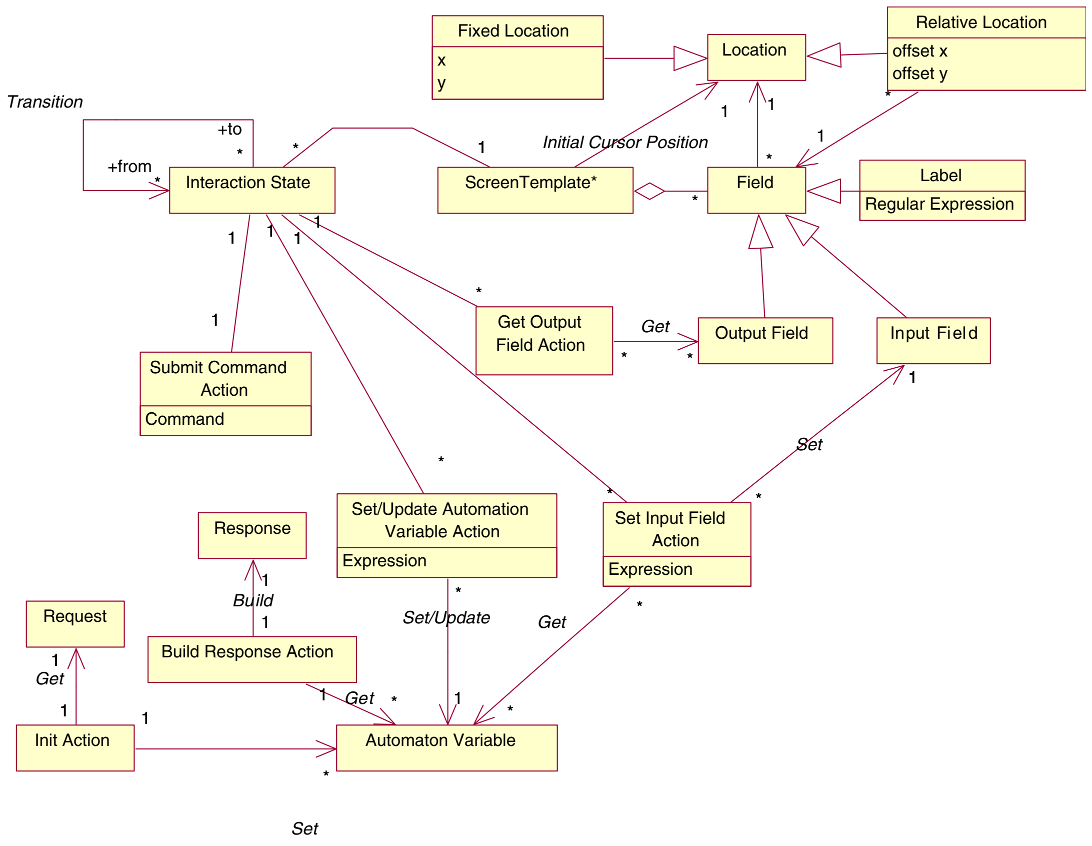


Fig. 10. Automaton description information model.

organizations analysis of their systems to find candidate services are needed. A business value and technical quality assessment of the candidate software assets will be needed in order to select possible candidates for the migration.

Some interesting proposals about how to solve this decision problem have recently been described in the literature. As an example, Sneed (2006) indicated a number of factors to be taken into account for the identification of candidate services in existing systems, like the reusability of the functionality, its business value (evaluated by a cost analysis taking into account the development, maintenance and redevelopment costs, as well as the return on investment), the degree of granularity of the resulting service, its state independence, and so on. Moreover, Lewis et al. (2006) from the SEI have proposed the SMART (Service-Oriented Migration and Reuse Technique) approach for gathering the needed information and identifying the risks of a migration effort in a systematic way.

5.2. Wrapping the selected use cases

In the second phase of the migration process, the selected use case will have to be wrapped. In order to use the wrapping technique proposed in this paper, three basic activities will have to be performed, namely S*ervice Identification*, *Reverse Engineering*, and *Wrapper Design*.

1. To maximise the reusability of the wrapped services, it is important for the selected use case to provide a functional abstraction with a high cohesion (Yourdon and Constantine, 1979). Therefore, Service Identification is an analysis activity that will be performed to establish whether the candidate use case can be transformed into a single, high cohesion service or, vice-versa, needs to be broken down into more elementary use cases, each of which can be wrapped into a more cohesive service. In the latter case, the selected use case will not be presented as a simple service but as a composite one obtainable by coordinating the execution flow of other services as a workflow of services (Hollingsworth, 1995). As an example, a use case that allows a user to browse a list of hotels and to book a room in a hotel from the list can be broken down into two interdependent use cases (browsing a list of hotels, and booking a hotel room), that need to be executed sequentially since the first one produces the output needed as input to the successive use case. In these cases, the original use case cannot be translated into a single service, but must be provided as two separate services that will be coordinated by means of a workflow. Of course, both wrapping solutions are characterised by specific strengths and weaknesses that will be analysed in the following Section 6.2 with reference to a case study.
2. The *Reverse engineering* activity will be executed to obtain a sufficiently comprehensive model of the interaction between the user and the legacy system for each candidate use case. To reach this goal, black-box techniques of dynamic analysis aiming at exercising the use case will be used to discover each alternative flow of actions (i.e., alternative scenario) of the use case. During the dynamic analysis, the sequence of screens returned by the system and exchanged messages must be collected, too, and classified in order to obtain the set of screen types and message types needed for building the Automaton.
   Due to the black-box reverse engineering approach, the completeness of the obtained interaction model cannot be determined, because of the risk of undiscovered interactions. As an example, many software systems present interaction scenarios that are triggered only by special input values (possibly used by the developer for debugging purposes). Such particular scenarios may be discovered only by analysing other sources of information, such as user guides or other development documentation about the legacy system (if it is available), or thanks to an accurate wrapper validation activity (see Section 5.3).
   A first output of this Reverse Engineering activity will be the Finite State Automaton, including the sets of interaction states, transitions, and actions triggering the transitions. This Automaton will be non-deterministic because, given a starting state and an input action, there will be more than one transition, each one reaching a different successive state.A second output will be the set of Screen Templates associated with each interaction state of the automaton, each including all the screen features that are needed for the automatic identification of the current interaction state implemented by the State Identifier component.
   The third output of this activity will be the interface of the wrapped service, including the set of input variables provided to the legacy system by the service request message, and the set of output variables that will be included in the service response message.
3. The goal of the *Wrapper Design* activity is to make the Interaction Model interpretable by the Automaton Engine. To reach this aim, the Interaction Model must be enriched with other information such as the data described in the Automaton description information model reported in Fig. 10. As an example, all the automaton variables needed to store the intermediate results of a use case execution must be defined, and each Interaction State of the FSA must be associated with the set of actions to be performed by the automaton engine either on these variables or on the screen fields. Finally, the information model must be translated in XML format and stored in the Repository.

5.3. Deploying and validating the wrapped use case

In the third phase of the migration process, all the operations needed to publish the service and export it to an Application Server have to be performed. This operation will depend on the technology adopted for implementing the service and the SOA, and is generally executable using specific commercial or open-source environments for service development and deployment. As an example, if the service must be exposed as a Web Service, the WSDL document (World Wide Web Consortium, 2001) describing input data contained in request messages and output data contained in response messages will have to be written and stored in an Application Server, while a UDDI description document (OASIS, 2004) of the service can be registered in a public UDDI repository.

After the deployment of the service on the application server, it will be possible to submit the wrapped functionality to a validation activity in its final execution environment. This validation will actually be a system test phase, aiming at discovering failures during the service execution due to Automaton design defects. Possible failures may consist of State Identifier exceptions (i.e., corresponding to unidentified screens) or an unexpected response produced by the service (i.e., the returned response message is different from the response that the legacy system user expects).

In order to maximise the effectiveness of this testing activity, different and complementary strategies will be used to design test cases. A first strategy will design test cases using the same input data that were used in the Reverse Engineering phase for modelling the legacy system user interface: this testing can be classified as regression testing, where the oracle for the tests will be provided by the original legacy system. A possible coverage criterion will require each use case scenario to be exercised at least once. To execute these tests it must be possible to set the pre-conditions of each scenario, and this may require adequate control of the state of the legacy system.

A second testing strategy may be based on analysis of the FSA model and will design test cases in order to exercise each independent path of the automaton, or all its nodes or edges. Also in this case, it may be necessary to control the internal state of the legacy system to set the pre-conditions for each test case.

6. Evaluation

The migration process presented in this paper has been applied in a series of case studies to identify its strengths and possible weaknesses. In this Section, the results of two migration case studies will be presented.

The subject system was the 'Pine' application (ver. 4.64) (University of Washington, 2006), a well-known client mail application that allows a user to read, compose and manage e-mail messages from an existing message box. Pine is a form-based legacy system based on stream-oriented terminals. The version of Pine used in the experiment is accessible via the Telnet protocol. Pine was selected for this experiment since it is a representative example of interactive form-based legacy systems, offering common and reusable functions for the management of a message box, which can therefore be considered potential candidates for the migration towards a SOA.

In the first case study, a fine-grained use case of Pine was transformed into a single Web Service, while in the second one a coarse-grained use case was migrated using two different wrapping strategies.

To execute both case studies the wrapper had to be implemented and deployed on an application server: all the components of the wrapper architecture were developed using Java, and the Apache Tomcat servlet container was used as the application server.[1]

6.1. First case study

In accordance with the migration process proposed in Section 5, the use cases offered by Pine had to be preliminarily analysed, and Web Services candidates to be selected among them. This selection aimed to maximise the reusability of the candidate services. Among the various use cases offered by Pine, we decided to migrate the *Get Message* use case that provides a basic and essential functionality of a client mail application and, therefore, represents a good candidate for a reusable service. *Get Message* allows the owner of a mailbox to get the text of a specific e-mail message contained in a given e-mail folder. To exercise this functionality, the user provides the system with her/his login and password, the name of the folder to access and the number of the message to read.

This use case presents seven different scenarios: three of them correspond to a successful message reading (distinguishing among one-page, two pages or more-pages message reading), while four of them correspond to failed message reading (caused either by incorrect user password, or empty folder, or inexistent folder, or inexistent message number).

To wrap this functionality, the Service Identification, Reverse Engineering, and Wrapper Design activities presented in Section 5.2 were performed by a software engineer who was familiar with the Pine application, and was an expert of the migration tasks to be accomplished since he belongs to the team of co-authors. Data about the effort spent on these tasks were recorded.

In the Service Identification phase, the software engineer established that the candidate use case showed functional cohesion and concluded that it could be migrated into a single service.

In the Reverse Engineering phase, a black-box analysis of the system was carried out in order to abstract the Finite State Automaton providing the Model of the Interaction with the legacy. The analysis was performed by running the system with different input data and conditions of the data persistently stored by the application, which reproduced all the possible interaction scenarios of the use case.

[1] The authors will distribute this tool to readers who may be interested in obtaining it for experimental purposes.

During this analysis, all the information needed to characterise the interactions was collected. At the end of the analysis, an FSA model including 23 interaction states and 28 transitions was produced. This automaton is reported in Fig. 11; it is characterised by 15 different Screen Templates that were associated with its interaction states. In this case, the number of Screen Templates was lower than the number of interaction states because some interaction states (such as states 17, 18, 19, 20 and 21) were characterised by identical Screen Templates.

The service interface included the following input data: user authentication data (*login* and *password*), name of the *folder* containing the message, and ordinal *number* of the requested message, while the output data included the following e-mail information: (*Date*, *From*, *To*, *Cc*, *Subject*, *Body*) or an *Exception message* if any error occurred.

The automaton was therefore verified by checking that each use case scenario was correctly associated with a path between the initial and final states of the automaton.

In the Design phase, the FSA model was completed by specifying the set of Automaton variables to be used as local buffers during the use case execution, and the actions to be performed by the automaton engine when it reaches each interaction state. The actions to be performed in the Initial and Final States were also designed. This model was translated into XML language and stored in the Automaton Repository.

A description of the Automaton is shown in Table 1. The first two columns in Table 1 report the identifier of the Interaction State and its description. The third column lists the Actions that will be executed by the engine when the corresponding State has been reached. The fourth column reports the Submit Command Action that will activate a State Transition. The last column provides the list of the possible states that may be reached from that state.

As shown in the Table, the Automaton includes five states with more than one possible consecutive state (i.e., states with numbers 2, 5, 9, 13, 14).

As to the automaton variables, 11 variables for storing the four fields of the Request message (*Login*, *Password*, *Folder*, *Message Number*) and the seven fields of the Response message (*Date*, *From*, *To*, *Cc*, *Subject*, *Body*, *Exception*) were defined.

After the Design phase, the software engineer had to execute the operations needed for deploying and validating the wrapped use case. The wrapper was preliminarily installed on an Application Server (the Apache Tomcat servlet container, together with Apache Axis libraries were used) and the FSA XML files were stored in the wrapper Repository. Moreover, the service interface was translated into a WSDL file.

The service was thereupon submitted to a validation activity in its execution environment. A client application invoking the service and providing it with input data was used to execute the validation. Test cases were designed for covering the seven scenarios of the *GetMessage* use case. Table 2 reports these test cases, with the description of the covered scenario and the corresponding interaction state path on the FSA graph. This test suite also covered the set of linear independent paths of the FSA graph. Thanks to this analysis some Screen Templates and Automaton description faults were detected.

Moreover, an additional *acceptance test* was performed that allowed some additional failures (due to automaton faults) to be discovered. This testing activity involved some students from Naples University, who were asked to request the *GetMessage* service using the client application at different times a day for about two weeks. The effort spent on this activity was 20 person-hours. A first failure was observed when the service was invoked concurrently

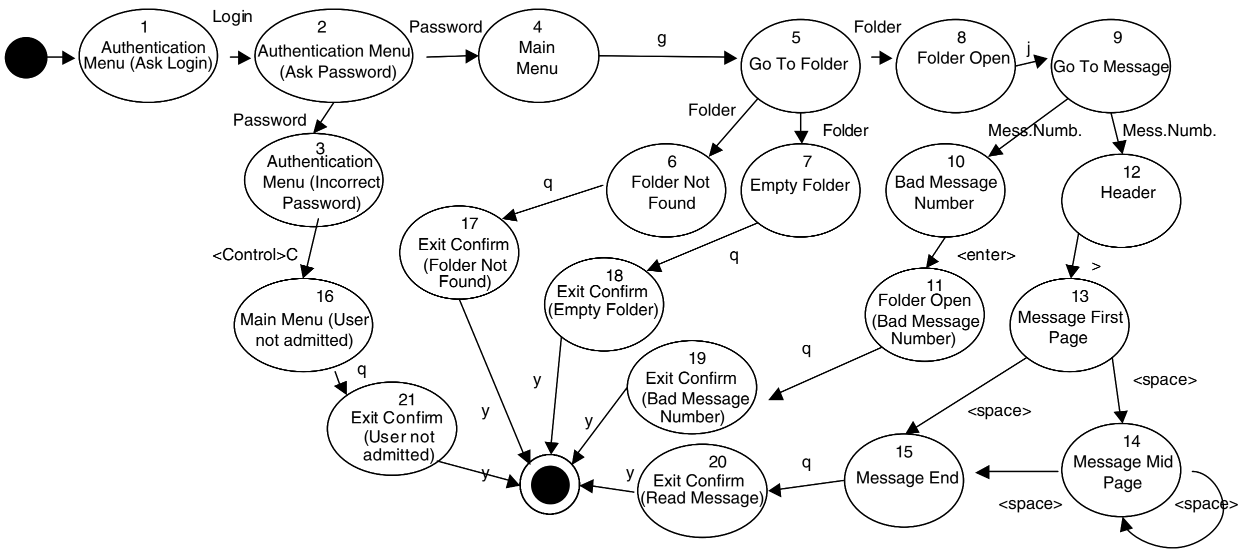

1 Authentication Menu (Ask Login)
Login
2 Authentication Menu (Ask Password)
Password
4 Main Menu
g
5 Go To Folder
Folder
8 Folder Open
j
9 Go To Message
Password
3 Authentication Menu (Incorrect Password)
Folder
Folder
6 Folder Not Found
7 Empty Folder
Mess.Numb.
Mess.Numb.
10 Bad Message Number
12 Header
q
17 Exit Confirm (Folder Not Found)
<Control>C
q
18 Exit Confirm (Empty Folder)
<enter>
>
11 Folder Open (Bad Message Number)
13 Message First Page
16 Main Menu (User not admitted)
q
19 Exit Confirm (Bad Message Number)
q
<space>
21 Exit Confirm (User not admitted)
y
y
y
<space>
15 Message End
14 Message Mid Page
y
y
20 Exit Confirm (Read Message)
q
<space>
<space>

Fig. 11. Graphic representation of the automaton of the "Get Message" use case.

Table 1
Automaton in the "Get-Message" use case

Interaction State ID	Interaction State Description	Actions	Submit Command	Next State
START		**Init** (*Login, Password, Folder, Message Number*)		1
1	Authentication Menu (Ask Login)	**Set Input Field**: *Login*	⟨Enter⟩	2
2	Authentication Menu (Ask Password)	**Set Input Field**: *Password*	⟨Enter⟩	3, 4
3	Authentication Menu (Incorrect Password)	**Set Automaton Variable**: *Exception* = "Incorrect Login and Password"	⟨Control⟩C	16
4	Main Menu		g	5
5	Go To Folder	**Set Input Field**: *Folder*	⟨Enter⟩	6, 7, 8
6	Folder Not Found	**Set Automaton Variable**: *Exception* = "Folder not found"	*q*	17
7	Empty Folder	**Set Automaton Variable**: *Exception* = "No messages in the folder"	*q*	18
8	Folder Open		j	9
9	Go To Message	**Set Input Field**: *Message Number*	⟨Enter⟩	10, 12
10	Bad Message Number	**Set Automaton Variable**: *Exception* = "Incorrect Message Number"	⟨Enter⟩	11
11	Folder Open (Message not found)		*q*	19
12	Header		>	13
13	Message First Page	**Get Output Fields**: (*Date, From, To, Cc, subject,, Body*); **Set Automaton Variables**: (Output: (*Date, From, To, Cc, subject, Body*))	⟨Space⟩	14, 15
14	Message Mid Page	**Get Output Field**: *Body*; **Update Automaton Variable**: *Body* = *Body* + Output:*Body*	⟨Space⟩	14, 15
15	Message End	**Get Output Field**: *Body*; **Update Automaton Variable**: *Body* = *Body* + Output:*Body*	*q*	20
16	Main Menu (User not admitted)		*q*	21
17	Exit Confirm (Folder Not Found)		*y*	END
18	Exit Confirm (Empty Folder)		*y*	END
19	Exit Confirm (No Message)		*y*	END
20	Exit Confirm (Read Message)		*y*	END
21	Exit Confirm (User not admitted)		*y*	END
END		**Build Response**: (*Date, From, To, Cc, Subject, Body, Exception*)		

Table 2
Test cases during the service validation activity

TC#	Covered scenario	Interaction State Path
1	One Page Message Reading	S-1-2-4-5-8-9-12-13-15-20-E
2	Two Pages Message Reading	S-1-2-4-5-6-9-12-13-14-15-20-E
3	More than Two Pages Message Reading	S-1-2-4-5-8-9-12-13-14-14-15-16-21-E
4	Bad Message Number	S-1-2-4-5-8-9-10-11-19-E
5	Empty Folder	S-1-2-4-5-7-18-E
6	Folder Not Found	S-1-2-4-5-6-17-E
7	Incorrect Password	S-1-2-3-16-17-E

by two client applications providing the same input request (involving the same user mailbox and folder). If the second request was received while another request was pending, the screen returned by the application was quite different from the one originally designed, since it contained an extra label notifying that the folder was open in read-only mode. In this case, a simple modification of a Screen Template was sufficient to solve the problem. A second failure was due to incompleteness of the automaton that did not include an interaction state (between state 2 and state 4) in which the system asked the user to choose between accepting or refusing a periodical maintenance intervention (this request is executed only at the first monthly activation of the application). This automaton fault was corrected by inserting the additional interaction state. After these modifications, a regression test was executed.

Effort data on the activities performed by the software engineer during the migration process are reported in Table 3. These data show that for migrating this use case, a software engineer spent 650 min (about 11 h) developing and validating the wrapped functionality. Most of this effort was spent in the Reverse Engineering and Validation tasks.

6.2. Second case study

The second case study involved a coarse-grained use case that was not directly provided by Pine but that was obtainable by executing several Pine functionalities sequentially: the use case implemented a message filtering functionality (named *Filter Messages* in the following) that move messages from a source folder to distinct destination folders according to a simple filtering rule.In particular, if the word *spam* is contained in the header of a message contained in the folder, then the message is moved to a folder named Spam, otherwise the message is moved to a folder named Good. This functionality replicates common user behaviour and also a facility offered by other client mail applications but that is not available in Pine.

This use case is coarse-grained since it can be broken down into two different use cases, the first one (*Get Headers*)

Table 3
Effort data on the *GetMessage* migration process

Phase	Activity	Activity effort (time)
Wrapping the use case	Service Identification	15 min
Wrapping the use case	Reverse Engineering of the FSA	200 min
Wrapping the use case	Reverse Engineering of the Screen Templates	90 min
Wrapping the use case	Wrapper Design: FSA Description Documents production	45 min
Deploy and Validation	Deploy and regression testing	300 min
Total effort		650 min

Table 4
Second case study automata characteristics

	Filter Messages (single Automaton)	Get Headers	Move Message
Use Case Scenario #	12	4	6
Interaction State #	36	12	32
Transition #	44	15	26
Different Screen template #	21	8	20

offering a user the possibility of getting the message headers of a given folder and the second one (*Move Messages*) for moving messages from a source folder to a destination one.

For wrapping coarse-grained use cases, two alternative options can be considered. A first solution consists of designing a single Automaton for executing the complete interaction with the legacy system needed to implement the coarse-grained service. The second solution is based on wrapping the component use cases and provides a coarse-grained service using a workflow of activities and of the more elementary services.

We decided to implement both solutions and collected data about both development activities in order to compare them. These migration activities were performed by two distinct members of the team of co-authors, who were experts on migration process and were familiar with the Pine application.

To implement the first solution, a wrapper able to implement the interaction that a user normally executes with the application to filter his messages had to be designed. This interaction exploits the Pine functionalities of *Get Headers* and *Move Messages* and includes the following steps: the user enters its mailbox (using her/his login and password), accesses a given folder, iteratively *Gets Headers* of the folder messages, and depending on the presence of the word spam in each header s/he will have to *Move Messages* either to the first or to the second destination folder. Therefore, the input data of this functionality are the login and password of the mailbox owner and the folder to filter. The only output is the number count of processed messages.

The *Filter Messages* use case presented 12 alternative scenarios. The Automaton implementing this use case was characterised by 36 Interaction States, 44 Transitions and 21 Screen Templates. The second column of Table 4 lists these data about the *Filter Messages* Automaton, while statistics regarding the effort spent on the corresponding migration process are reported in the second column of Table 5. The total effort required to develop this first solution was 1165 person-minutes.

The second wrapping solution implemented the new service by a workflow of services, obtained by wrapping reusable functionalities of the legacy application. To develop this solution, the second software engineer had to preliminarily identify the reusable functionalities *Get Headers* and *Move Messages* and wrap them as services, then he had to develop a workflow implementing the *Filter Messages* use case by reusing these services, and to deploy the final composite service.

The characteristics of the automata providing *Get Headers* and *Move Messages* services are reported in the third and fourth columns in Table 4, while the third and fourth columns in Table 5 show the corresponding migration process effort data. The workflow that implements this composite service has been reported in Fig. 12 as a UML Activity Diagram. In the case study, the workflow was specified as a BPEL workflow (OASIS, 2005), and it was deployed on an application server including Active BPEL, a runtime environment that is capable of executing BPEL process definitions (Active BPEL, 2006). Finally, a testing activity aiming at exercising the 12 scenarios of the use case was performed. The effort data (expressed in minutes) spent by the software engineer on designing, deploying and testing this workflow are reported in Table 6. The total effort required to develop this second solution was 1545 person-minutes.

Data from this case study were used to compare the wrapping solutions in terms of wrapping design and implementation efforts, reusability and flexibility of obtained ser-

Table 5
Service migration effort data from the second case study (all values in minutes)

Migration Activity	Filter Messages (single automaton)	Get Headers	Move Message
Service Identification	25	10	15
Reverse Engineering of the FSA	360	120	240
Reverse Engineering of the Screen Templates	120	70	120
Wrapper Design: FSA Description Documents production	60	30	60
Wrapper Deploy and regression testing	600	150	450
Total	1165	380	885

vices, and service performance. The results of this comparison are analysed and interpreted in the following:

1. As far as wrapping design and implementation efforts are concerned, the solution with a unique automaton was less expensive (1165 person-minutes) than the solution based on the workflow of services (1545 person-minutes, of which 280 were devoted to the workflow and 1265 to the component Web Services). This datum depended on the need to develop the *Get Headers* and *Move Messages* services from scratch so as to implement the second solution. Of course, if the composing services had already been available, the effort required for implementing just the workflow (see Table 6) would have been sensibly lower. These effort data showed that the migration costs were low and acceptable both for fine-grained and coarse-grained use cases, but of course the effort grows with the functional complexity of the use case. Moreover, most of the effort is always devoted to the Reverse Engineering tasks performed for Screen Template and FSA models abstraction, and to the wrapper validation task.
2. As to the reusability of the solutions, the workflow-based solution provided more reusable services (i.e., *Filter Messages*, *Get Headers*, and *Move Messages*), than the first one (it just provides the reusable service *Filter Messages*). This datum confirms that service reusability depends on the degree of cohesion of its functionality: the more cohesive the functionality, the more reusable the service (Sneed, 2006).

Table 6
Workflow development effort data

Workflow Design	60 min
BPEL document production	20 min
Deploy and validation	200 min
Total	280 min

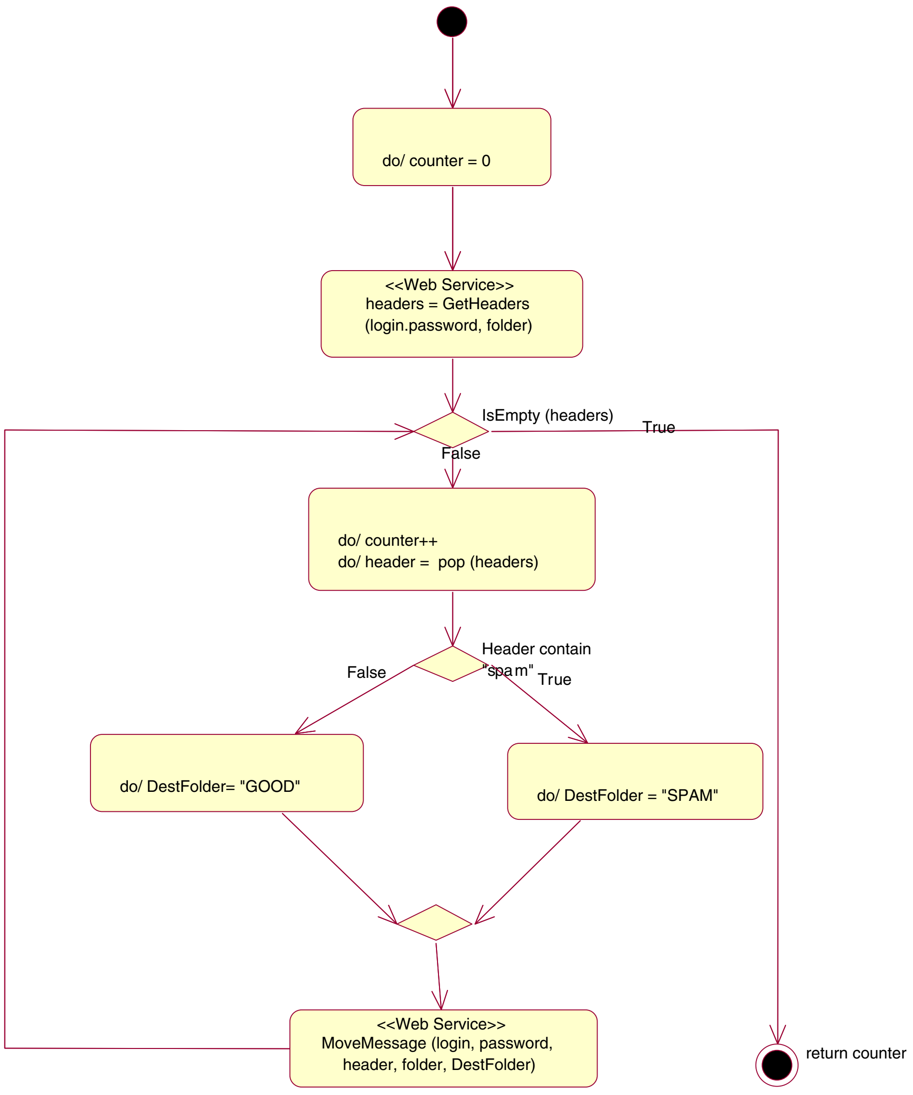


Fig. 12. Workflow modelling the composite Web Service *Filter Messages*.

3. As to the flexibility of the resulting service, the workflow-based solution is more adaptable than the first one, since the use case logic is not embedded in an automaton, but in the workflow. A modification involving this control logic will impact just the workflow, leaving the services unaffected.
4. As to the performance, the response times of the two solutions to the same service request were compared: it was observed that the solution based on a unique automaton showed a better response time than the workflow-based one. This datum was due to the fact the wrapper execution of the single automaton involved a lower number of interaction states that had to be visited during the interpretation, and required only a single instance of the legacy system execution to be launched. Vice-versa, the second solution required more instances of legacy system execution (one for each Web Service invocation) and a greater number of interaction states had to be visited. However, it should be noted that a workflow execution time depends on the workflow design, too: the response time may be improved by executing some activities concurrently, but this solution could not be applied in the case study, due to the message locking policy applied by Pine.

In conclusion, this preliminary experiment showed us the feasibility and cost-effectiveness of the proposed migration approach. Of course, the reduced number of form-based systems involved in the experiment (just one system), the reduced number of migration projects (three cases) carried out, the familiarity of the software engineers with the Pine application, their knowledge of the migration process and methods and technologies used during the migration project, are among, but do not exhaust the list of possible threats to the validity of this experimental result. To address these and other possible threats, an empirical validation of this approach (Zelkowitz and Wallace, 1998) consisting of replicated migration experiments, involving different subjects software systems and software engineers with distinct skills and experience, will be needed.

7. Conclusions and future works

Software systems modernisation using Service Oriented Architectures and Web Services represents a valuable option for extending the lifetime of mission-critical legacy systems. Exposing them as services allows heterogeneous systems to become interconnected and interoperable, and can be implemented by wrapping techniques that provide a low risk short-term solution, promising a quick turnaround and minimal changes to the existing legacy platform.

Of course, methods, techniques and tools are needed for migrating a legacy system towards a SOA effectively and efficiently. In this paper, we have presented a novel black-box wrapping technique for exporting the functionalities implemented by interactive form-based systems towards Service Oriented Architectures. With respect to other wrapping techniques proposed in the literature, this approach is novel because it wraps the whole legacy system, rather than specific and limited parts of its code, and is not invasive to the legacy system, whose configuration and execution environment are left untouched.

A further point of novelty of this approach consists of the idea of implementing the wrapper as an interpreter of a Finite State Automaton that encapsulates the rules of the interaction between the user and the legacy system. Thanks to this interpretation, the wrapper can eliminate the user intervention required to obtain processing from a form-based system and exposes this processing as a Service. The Finite State Automaton needed by the wrapper can be reverse engineered from the legacy system using dynamic analysis techniques.

The paper also proposed a modular architecture of the wrapper and the steps of a migration process defining the flow of activities to be carried out for wrapping legacy system functionalities. For validating our approach we developed the wrapper using Java and XML technologies, and conducted an experiment consisting of some migration case studies whose results have been presented in the paper.

The case studies aimed at exploring the feasibility and cost-effectiveness of the migration process: effort data were collected and analysed. In the first case study, a fine-grained use case was transformed into a single Web Service, while in the second one two different wrapping strategies were used to migrate a coarse-grained use case.

Both case studies showed the cost-effectiveness of the approach and, however, highlighted the need to devise methods and techniques for supporting the process tasks of Automaton reverse engineering and validation.

Moreover, the second case study evaluated the possibility of combining the wrapping approach with the workflow of services technology. The experimental results showed that our migration technique can be successful adopted in combination with workflow technologies to obtain more reusable and flexible services.

This preliminary experiment showed us the feasibility of the migration approach, and indicated directions for future works. In particular, techniques and tools for aiding the most expensive tasks of the migration process will have to be developed in order to improve the scalability of the approach. Moreover, empirical studies will have to be designed and executed to extend the validity of the case studies findings presented in this paper. In particular, one of these further experiments will assess the transferability of the approach to external organizations and the QoS (Quality of Service) of the wrapped services.

References

Active BPEL, 2006. Active BPEL Engine Technology. Available from: <http://www.activebpel.org/index.html>.

Atkins, D.L., Ball, T., Bruns, G., Cox, K., 1999. Mawl: a domain-specific language for form-based services. IEEE Transactions on Software Engineering 25 (3), 334–346.

Baumgartner, R., Gottlob, G., Herzog, M., Slany, W., 2004. Interactively adding Web service interfaces to existing Web applications. In: Proceedings of the International Symposium on Applications and the Internet, pp. 74–80.

Bennett, K., 1995. Legacy systems: coping with stress. IEEE Software 12 (1), 19–23.

Berstel, J., Crespi Reghizzi, S., Roussel, G., San Pietro, P., 2005. A scalable formal method for design and automatic checking of user interfaces. ACM Transactions on Software Engineering and Methodology (TOSEM) 14 (2), 124–167.

Bi, Y., Hull, M.E.C., Nicholl, P.N., 2002. An XML approach for legacy code reuse. Journal of Systems and Software 61, 77–89.

Bisbal, J., Lawless, D., Wu, B., Grimson, J., 1999. Legacy information systems: issues and directions. IEEE Software 16 (5), 103–111.

Bovenzi, D., Canfora, G., Fasolino, A.R., 2003. Enabling legacy system accessibility by Web heterogeneous clients. Proceedings of the 7th European Conference on Software Maintenance and Reengineering, CSMR 2003. IEEE CS Press, pp. 73–81.

Canfora, G., Fasolino, A.R., Frattolillo, G., Tramontana, P., 2006. Migrating interactive legacy system to Web services. Proceedings of the 10th European Conference on Software Maintenance and Reengineering. IEEE CS Press, pp. 23–32.

Chan, K., Liang, Z.C.L., Michail, A., 2003. Design recovery of interactive graphical applications. Proceedings of the 25th International Conference on Software Engineering, ICSE 2003. IEEE CS Press, pp. 114–124.

Comella-Dorda, S., Wallnau, K., Seacord, R.C., Robert, J., 2000. A survey of black-box modernisation approaches for information systems. Proceedings of the International Conference on Software Maintenance. IEEE CS Press, pp. 173–183.

De Lucia, A., Fasolino, A.R., Pompella, E., 2001. A decisional framework for legacy system management. Proceedings of the IEEE International Conference on Software Maintenance. IEEE CS Press, pp. 642–651.

Draheim, D., Weber, G., 2005. Form-Oriented Analysis. In: A New Methodology to Model Form-Based Applications. Springer-Verlag.

Draheim, D., Lutteroth, C., Weber, G., 2005. A source code independent reverse engineering tool for dynamic Web sites. Proceedings of 9th European Conference on Software Maintenance and Reengineering, CSMR 2005. IEEE CS Press, pp. 168–177.

Gaeremynck, Y., Bergman, L.D., Lau, T., 2003. MORE for less: model recovery from visual interfaces for multi-device application design. Proceedings of the 8th ACM International Conference on Intelligent User Interfaces IUI'2003. ACM Press, pp. 69–76.

Guo, H., Guo, C., Chen, F., Yang, H., 2005. Wrapping client–server application to Web services for Internet computing. Proceedings of the Sixth International Conference on Parallel and Distributed Computing, Applications and Technologies, PDCAT 2005. IEEE CS Press, pp. 366–370.

Hollingsworth, D., 1995. Workflow Management Coalition: The Workflow Reference Model. Available from: <http://www.wfmc.org/standards/docs/tc003v11.pdf>.

IEEE 1998. IEEE Std. 1219-1998, Standard for Software Maintenance. IEEE CS Press, Los Alamitos, CA.

Jagacy Software, 2006. Available from: <http://www.jagacy.com>.

Jiang, Y., Stroulia, E., 2004. Towards reengineering Web sites to Web-services providers. Proceedings of the 8th European Conference on Software Maintenance and Reengineering, CSMR 2004. IEEE CS Press, pp. 296–305.

Lewis, G., Morris, E., Smith, D., 2006. Analyzing the reuse potential of migrating legacy components to a service-oriented architecture. Proceedings of the 10th European Conference on Software Maintenance and Reengineering. IEEE CS Press, pp. 14–22.

Litoiu, M., 2004. Migrating to Web services: a performance engineering approach. Journal of Software Maintenance and Evolution: Research and Practice 16 (1–2), 51–70.

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

Nartovich, A., Henkel, D., O'Shannessy, S., Owen, B., 2004. The IBM WebFacing Tool: Converting 5250 Applications to Browser-based GUIs. IBM Redbooks. Available from: <http://www.redbooks.ibm.com/redbooks/pdfs/sg246801.pdf>.

Novell exteNd Composer Enterprise Edition 5.2, 2006. Available from: <http://www.novell.com/products/extend/composer/overview.html>.

OASIS, 2004. UDDI Spec Technical Committee Draft Version 3.0.2. Available from: <http://uddi.org/pubs/uddi-v3.0.2-20041019.htm>.

OASIS, 2005. Web Services Business Process Execution Language Version 2.0. Available from: <http://www.oasis-open.org/committees/download.php/16024/wsbpel-specification-draft-Dec-22-2005.htm>.

OASIS, 2006. Reference Model for Service Oriented Architecture 1.0. Committee Specification 1, 2 August 2006. Available from: <http://www.oasis-open.org/committees/download.php/19679/soa-rm-cs.pdf>.

Olsen, D.R., 1984. Pushdown automata for user interface management. Transactions on Graphics (TOG) 3 (3), 177–203.

Parnas, D.L., 1969. On the use of transition diagrams in the design of a User Interface for an interactive computer system. Proceedings of the 24th National Conference. ACM Press, pp. 379–385.

Rodriguez, J.R., Belapurka, V., Carvalho, R., Filomeno, A., Queixas, W., Quixchan, O., 2001. A Comprehensive Guide to IBM WebSphere Host Publisher Version 3.5. IBM Redbook. Available from: <http://www.redbooks.ibm.com/redbooks/pdfs/sg246281.pdf>.

Rugaber, S., 1999. A tool suite for evolving legacy software. Proceedings of IEEE International Conference on Software Maintenance, ICSM'99. IEEE CS Press, pp. 33–39.

Sipser, M., 1997. Introduction to the Theory of Computation. PWS, Boston, Massachusetts, Section 1.2: Nondeterminism, pp. 47–63.

Sneed, H.M., 2001. Wrapping legacy COBOL programs behind an XML-interface. Proceedings of 8th Working Conference on Reverse Engineering, WCRE 2001. IEEE CS Press, pp. 189–197.

Sneed, H., 2006. Integrating legacy software into a Service Oriented Architecture. Proceedings of the 10th European Conference on Software Maintenance and Reengineering. IEEE CS Press, pp. 3–13.

Sneed, H.M., Sneed, S.H., 2003. Creating Web services from legacy host programs. Proceedings of the 5th IEEE International Workshop on Web Site Evolution, WSE 2003. IEEE CS Press, pp. 59–65.

Stroulia, E., El-Ramly, M., Kong, L., Sorenson, P., Matichuk, B., 1999. Reverse engineering legacy interfaces: an interaction-driven approach. Proceedings of the Sixth IEEE Working Conference on Reverse Engineering. IEEE CS Press, pp. 292–302.

Stroulia, E., El-Ramly, M., Sorenson, P., 2002. From legacy to Web through interaction modelling. Proceedings of International Conference on Software Maintenance. IEEE CS Press, pp. 320–329.

Tilley, S., Gerdes, J., Hamilton, T., Huang, S., Muller, H., Wong, K., 2002. Adoption challenges in migrating to Web services. Proceedings of the 4th International Workshop on Web Site Evolution, WSE 2002. IEEE CS Press, pp. 21–29.

University of Washington, 2006. Pine® – a Program for Internet News & Email. Available from: <http://www.washington.edu/pine/index.html>.

Wasserman, A.I., 1985. Extending state transition diagrams for the specification of human–computer interaction. IEEE Transaction on Software Engineering 11 (8), 699–713.

World Wide Web Consortium, 2001. Web Service Description Language (WSDL) 1.1. Available from: <http://www.w3.org/TR/wsdl>.

World Wide Web Consortium, 2003. Simple Object Access Protocol (SOAP) 1.2. Available from: <http://www.w3.org/TR/soap/>.

World Wide Web Consortium, 2004. Web Services Architecture. Available from: <http://www.w3.org/TR/ws-arch/>.

Yourdon, E., Constantine, L.L., 1979. Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design. Prentice-Hall.

Zdun, U., 2002. Reengineering to the Web: a reference architecture. Proceedings of IEEE European Conference on Software Maintenance and Reengineering. IEEE CS Press, pp. 164–173.

Zelkowitz, M.V., Wallace, D.R., 1998. Experimental models for validating technology. IEEE Computer 31 (5), 23–31.

Zhang, Z., Yang, H., 2004. Incubating services in legacy systems for architectural migration. Proceedings of the 11th Asia-Pacific Software Engineering Conference. IEEE CS Press, pp. 196–203.

Gerardo Canfora is a full professor of computer science at the Faculty of Engineering and is the Director of the Research Centre on Software Technology (RCOST) of the University of Sannio in Benevento, Italy. He serves on the program committees of a number of international conferences. He was a program co-chair at the 1997 International Workshop on Program Comprehension, at the 2001 International Conference on Software Maintenance, the 2004 European Conference on Software Maintenance and Reengineering, the 2005 International Workshop on principles of Software Evolution, and the 2007 International Conference on Software Maintenance. He was the General chair at the 2003 European Conference on Software Maintenance and Reengineering and the 2004 Working Conference on Reverse Engineering. His research interests include software maintenance, program comprehension, reverse engineering workflow management, metrics, and experimental software engineering. He serves on the Editorial Board of The Journal of Software Maintenance and Evolution. He is a member of the IEEE and the IEEE Computer Society.

Anna Rita Fasolino is an Associate Professor of Computer Science at the University of Naples "Federico II", Italy. From 1998 to 1999 she was at the Computer Science Department of the University of Bari, Italy. She serves on the program committees of a number of international conferences. Her research interests include software maintenance and quality, reverse engineering, Web engineering and Web testing. She was awarded her degree in Electronic Engineering (summa cum laude) and her Ph.D. in Electronic Engineering and Computer Science by the University of Naples Federico II, Italy.

Gianni Frattolillo, is an Independent ICT Engineering Professional working in Italy. Formerly, he worked as an IT Architect at an IBM Company. He was awarded his degree in Computer Engineering by the University of Naples "Federico II". His present interests are focused on the fields of software engineering, project management and ICT security. He writes for some Italian ICT magazines. He is a member of the Project Management Institute (PMI) and the PMI Southern Italy Chapter (PMI-SIC).

Porfirio Tramontana, is an Assistant Professor of Computer Science at the University of Naples "Federico II", Italy, where he holds a professorship in Software Engineering. He was awarded his degree and then his Ph.D. in Computer Engineering, by the University of Naples "Federico II". His present research interests are focused on the fields of software maintenance, reverse engineering, software comprehension, Web engineering, and reengineering.