Creating Web Services From GUI-Based Applications

Mark Grechanik, Kevin M. Conroy, and Kishore S. Swaminathan
Systems Integration Group, Accenture Technology Labs
Chicago, IL 60601
Email: {mark.grechanik, kevin.m.conroy, k.s.swaminathan}@accenture.com

***Abstract*— *Graphical User Interface (GUI) APplications (GAPs)* are ubiquitous and provide various services. However, it is difficult to make GAPs exchange information (i.e., interoperate) especially if they are closed and monolithic. Unlike GAPs, web services are applications that are designed to interoperate over the Internet. Thus a fundamental problem of interoperability is how to reengineer GAPs into web services efficiently and non-invasively.**

We propose a novel generic approach for creating web services from GAPs. This approach combines a nonstandard use of accessibility technologies for accessing and controlling GAPs in a uniform way with a visualization mechanism that enables nonprogrammers to create web services by performing point-and-click, drag-and-drop operations on GAPs. We built a tool based on our approach and created web services that control two closed and monolithic commercial GAPs with this tool. Our evaluation suggests that our approach is effective and it can be used to create web services from nontrivial GAPs.

I. Introduction

Organizations custom-build *Graphical User Interface (GUI) APplications (GAPs)* and acquire them from third-party vendors to assist in business operations. GAPs are ubiquitous and provide various services, and they are often required to exchange information (i.e., to interoperate [5]) in order to improve the quality and to extend the coverage of their services. However, it is difficult to interoperate GAPs because many of them are closed and monolithic, and they do not expose any programming interfaces or data in known formats.

Web services are software components that interoperate over the Internet, and they gain widespread acceptance partly because of the business demand for applications to exchange information [7]. Unlike GAPs, using web services enables organizations to automate business processes by increasing the speed and effectiveness of information exchange. Naturally, different organizations explore how to replace legacy GAPs with web services so that these organizations can improve their business processes by interoperating web services.

However, replacing legacy GAPs with web services is difficult for most organizations that have invested heavily in a variety of GAPs from multiple vendors [1]. Given the complexity of GAPs and the cost of replacing them, reusing GAPs can ease transitioning to web services. Thus, a fundamental problem of interoperability is how to reengineer GAPs into web services efficiently, so that users can access the services of these GAPs programmatically.

Reengineering GAPs into web services is difficult because of brittle legacy architectures, poor documentation, significant programming effort, and subsequently, the large cost of these projects. This situation is aggravated by the fact that businesses use successful GAPs for decades, and managers are understandably reluctant to authorize changes to source code as any modifications to GAPs may break them and disrupt well-established business services. Converting third-party GAPs into web services may not even be possible since organizations often do not have access to the source code. Therefore, it is desirable to reengineer GAPs into web services non-invasively and efficiently.

Our main contribution is a novel generic approach for creating web services from GAPs. This approach combines a nonstandard use of accessibility technologies for accessing and controlling GAPs in a uniform way with a visualization mechanism that enables nonprogrammers to create web services by performing point-and-click, drag-and-drop operations against closed and monolithic GAPs. Since accessibility technologies are present on major computing platforms to allow disabled users to access applications, we utilize these technologies in our uniform mechanism of creating web services from GAPs.

A fundamental difference between our approach and related work is that we offer a generic and noninvasive mechanism that enables nonprogrammers to create web services from GAPs using intuitive visual interfaces without modifying any source code and without making any changes to the underlying computing platforms.

In addition, we evaluate the performance of controlling GAPs using their GUI interfaces and compare it with the performance measurements of pure programmatic web services. Based on these evaluation results, we provide a framework for recommending when web services should be written from scratch versus reusing legacy GAPs. To our knowledge, this result has not been published before.

We built a tool based on our approach, and we used this tool to create web services from two closed and monolithic commercial GAPs. We describe our experience with this tool, measure and analyze performance characteristics of the created web services. The results suggest that our approach is efficient and effective.

II. Related Work

Related work falls into two categories: approaches for converting GAPs into software components of different types, and techniques used by these approaches to control and manipulate GAPs programmatically. Wrapper approaches are the closest related work from the former category since we can view web services as wrappers for the corresponding GAPs.

A result that is directly relevant to our approach is the CAWOM toolkit that enables programmers to create CORBA services from legacy command-line programs (CLPs), i.e., programs whose functionality is accessible only through command-line inputs [15]. In this case, we consider command-line as a degenerated form of GUI interface. The idea of CAWOM is to generate a wrapper around a CLP to programmatically invoke its commands. While our approach deals with creating web services from GAPs that have rich graphic elements, CAWOM is designed to create CORBA services only from CLPs by intercepting and extracting data from the command line input/output.

Wrapper Generator is a tool that wraps multithreaded, legacy code as a single CORBA object for use in a distributed component-based environment [10]. Unlike our approach, the Wrapper Generator depends upon parsing and transforming the source code of the applications into CORBA objects.

A web browser-shell integrates a command interpreter into the browser's location box to automate HTML interfaces [12]. A browser-shell wraps legacy CLPs with an HTML/CGI graphical interface. This approach is heavily dependent upon parsing HTML and extracting data from the command line input/output, and in that way it is significantly different from our approach which does not need to parse any source code.

Appletizing is an approach that transforms a Java GUI application into a Java applet running inside a web browser [14]. Unlike our approach, appletizing is heavily dependent on the Java bytecode format, which it uses to transform the original application's bytecode into a distributed application. In contrast, our approach does not rely on platform-specific data or code formats since it uses a generic accessibility layer present on all major computing platforms.

Code patching [6] and binary rewriting [9] techniques modify the binary code of executable programs in order to control and manipulate them when converting these programs into applications of different types, and these techniques are used to control and manipulate GAPs [8]. However, these techniques are platform-dependent, and programmers are required to write complicated code to change program executables. Using these techniques is difficult and error prone, and often causes applications to become unstable and crash.

When it comes to extracting information from GAPs and their GUI elements, the term *screen-scraping* summarily describes various techniques for automating user interfaces [13] [3]. Macro recorders use this technique by recording the users mouse movements and keystrokes, then playing them back by inserting simulated mouse and keyboard events in the system queue [11]. Screen-scraping has multiple advantages over binary rewriting and code patching techniques as it does not require any modifications to the underlying computing platforms or applications' source and executable code. Our approach uses some aspects of screen-scraping, however, it differs from other screen-scraping techniques since it does not depend on parsing a scripting language that describes the GUI, and therefore it is more generic and uniform.

There are few commercial tools available for transitioning legacy applications to web services. AttachmateWRQ and Seagull Software Corp. are the largest and oldest companies providing tools for migrating legacy applications to web services. For each platform they have separate lines of products that exploit ad-hoc platform-specific techniques for screen-scraping. These and other companies have built different tools for converting GAPs into web services for different platforms. Doing that results in multiple versions of the source code for these tools, subsequenlty their increased cost, and eventually difficulties in maintaining and evolving different codebases. In constrast, our approach uses a platform-independent mechanism for controlling and manipulating GAPs.

III. The Problem Statement

Our goal is to design an approach for creating web services from GAPs with a high-degree of automation. These web services should provide their functionality in large-scale, distributed enterprise environments by orchestrating different GAPs that are located on different computers and running on different platforms.

Our approach should be non-invasive, specifically it should not require users to modify GAPs or change the existing configuration of GAPs in an enterprise environment. Our approach should be easy to use so that nonprogrammers can create web services using their basic knowledge of how to interact with GAPs to accomplish business tasks. The wide applicability of our solution should be achieved by using an underlying technology for controlling and manipulating GAPs that is common to major computing platforms. The results of GAP computations initiated by web services should be identical to those obtained by users accessing and controlling GAPs through their GUIs.

This problem statement is based on real-world requirements. Large-scale distributed enterprise environments contain thousands of computers running dozens of different platforms and thousands of different applications. Any approach that would impose constraints on these environments is unrealistic. For example, requiring GAPs to run on the same computers with web services that use them may contradict a security policy that places web services and GAPs on different sides of the enterprise firewall.

IV. Our Solution

In this section, we present core ideas behind our approach, provide a background on accessibility technologies, and give a high-level overview of how our solution is used to create web services from GAPs.

A. Core Ideas

Our main idea is to create web services that mimic a human-driven procedure of interacting with GAPs. This procedure can be described as follows. After a GAP is started and initial screens appear, users may read data from and enter data into some GUI elements. Then users initiate transitions by causing some actions (e.g., select a menu item or click on a button). As a result of these actions, GAPs perform computations and show GUI screens that may be different from the previous ones. Again, users read and enter some data and perform actions. This cycle continues until users quit these applications.

In order for web services to mimic this procedure, they should be able to access GUI elements of GAPs programmatically. A core idea of our solution is that GAPs and their GUI elements are programming objects whose values can be set and retrieved and whose methods are associated with actions that users perform on these elements. For example, a combo box element displays a number of items, and selecting an item in the combo box invokes a method that performs some computation. To control this combo box programmatically, a service should invoke methods and set values of the fields of a programming object that represents this combo box which is hosted in a GUI of some GAP.

A key solution is to use GAPs as programming objects and GUI elements of these GAPs as fields of these objects, and to perform actions on these GUI elements by invoking methods on the objects that represent these GAPs. Unfortunately, services cannot access and manipulate GUI elements of GAPs as pure programming objects because GUI elements only support user-level interactions. Accessibility technologies overcome this limitation by exposing a special interface whose methods can be invoked and the values of whose fields can be set and retrieved thereby controlling GUI elements that have this interface. We give an overview of the accessibility technologies in the next Section IV-B.

B. Accessibility Technologies

Accessibility technologies provide different aids to disabled computer users [4]. Specific aids include screen readers for the visually impaired, visual indicators or captions for users with hearing loss, and software to compensate for motion disabilities. Most computing platforms include accessibility technologies since electronic and information technology products and services are required to meet the *Electronic and Information Accessibility Standards* [4]. For example, *Microsoft Active Accessibility (MSAA)* technology is designed to improve the way accessibility aids work with applications running on Windows, and *Sun Microsystems Accessibility* technology assists disabled users who run software on top of *Java Virtual Machine (JVM)*. Accessibility technologies are incorporated into these and other computing platforms as well as libraries and applications in order to expose information about user interface elements.

Accessibility technologies provide a wealth of sophisticated services required to retrieve attributes of GUI elements, set and retrieve their values, and generate and intercept different events. In this paper, we use MSAA for Windows, however, using a different accessibility technology will yield similar results. Even though there is no standard for accessibility *Application Programming Interface (API)* calls, different technologies offer similar API calls, suggesting slow convergence towards a common programming standard for accessibility technologies.

The main idea of most implementations of accessibility technologies is that GUI elements expose a well-known interface that exports methods for accessing and manipulating the properties and the behavior of these elements. For example, a Windows GUI element should implement the `IAccessible` interface in order to be accessed and controlled using the MSAA API calls. Programmers may write code to access and control GUI elements of GAPs as if these elements were standard programming objects.

Using accessibility technologies, programmers can also register callback functions for different events produced by GUI elements thereby obtaining timely information about states of the GUI elements of the GAPs. For example, if a GUI element receives an incorrect input and the GAP shows an error message dialog informing the user about the mistake, then a previously registered callback can intercept this event signaling that the message dialog is being created, dismiss it, and send an “illegal input” message to the web service controlling the GAP.

C. A Birds-Eye View

We present a birds-eye view of how our approach is used by giving an example of creating a web service for `Quicken Expensable 98`, a commercial Windows-based application that allows users to enter their expenses. `Quicken Expensable 98` is a closed and monolithic GAP that does not have any programming interfaces and stores its expense data in a proprietary binary format. There is no publicly available source code, and the application was designed only for user-level interactions.

We have built a tool called Designer that enables users to create web services from GAPs using our approach. The front end of Designer is shown in Figure 1. Using the Designer, a user enters the name of the desired web service (e.g., `QuickenExpensable`) and the name of the exported method of this service (e.g., `submitExpense`). The user also specifies that the service controls `Quicken Expensable 98` by providing its location to the Designer. This information is shown in the leftmost tab (i.e., `Service Explorer`) of the Designer.

Next, the user interacts with the GAP `Quicken Expensable 98` by performing actions against GUI elements to enter an expense. For example, the user selects an expense envelope on the first screen, and then double clicks on the entry in the envelope list box. These actions cause the GAP to switch to the expense entry screen. This and other screens of the GAP are shown in Figure 3 as part of the state machine diagram.

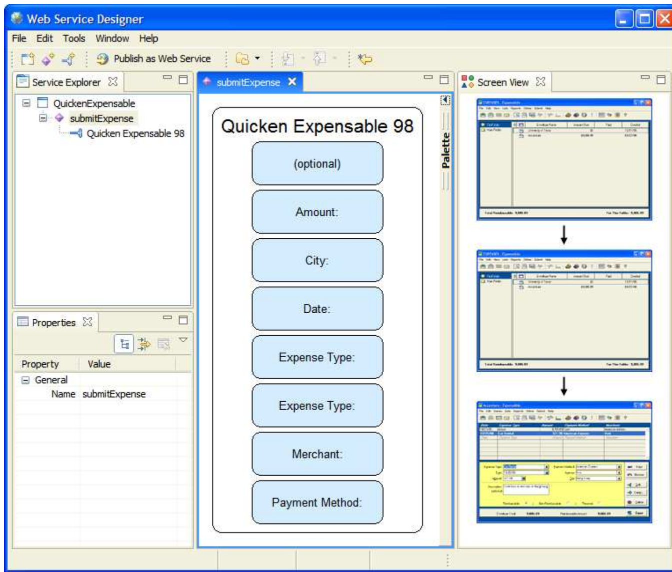

Fig. 1. The front-end of the Designer.

The purpose of interacting with the GAP is to allow the Designer to record the structures of the screens and user actions on the GAP and then transcode these actions into programming instructions that the resulting web service will execute. To do that, the Designer intercepts selected user-level events using the accessibility layer. These events allow the Designer to record the sequence of screens that the user goes through as well as the actions that the user performs on GUI elements in order to enter the expense. This sequence of screen is shown in the rightmost tab (i.e., `Screen View`) of the Designer which is shown in Figure 1.

When recording the sequence of screens, the Designer obtains information about the structure of the GUI and all properties of individual elements using the accessibility-enabled interfaces. This information allows the web service to locate GUI elements in order to set or retrieve their values or to perform actions on them in response to requests of its clients.

At the design time, the user should specify what GUI elements will receive the values of input parameters from the web service. For example, the user enters the payment method for an expense, (e.g., Visa) in the GUI element labeled `Payment Method`. This element should be mapped to the input parameter of the exposed method `submitExpense` of the web service. To do that, the user moves the cursor over the GUI element, and the Designer uses the accessibility API calls to obtain information about this element. To confirm the selection, a frame is drawn around the element with the tooltip window displaying the information about the selected element. Then, the user clicks the mouse button and drags this element (or rather its image) onto the middle tab of the Designer. After releasing the mouse button, the dragged element is dropped onto the input parameter palette of the Designer under the label `Quicken Expensable 98`.

Once the user has dragged-and-dropped all GUI elements that correspond to the input parameters of the method of the web service, the service can be published simply by clicking on the button `Publish as Web Service`. The Designer uses the information captured for each screen and input elements to generate the Java code for the web service, compiles it, and deploy it to a web services platform such as Apache Axis. When this service is called from a client, the method `submitExpense` takes input parameters that describe the expense and uses the accessibility interfaces to control and manipulate `Quicken Expensable 98` to enter the expense. A short movie demonstrating this procedure as well as the Designer, is available at our website [www.markgrechanik.com]. It can be viewed inside the browser [http://www.markgrechanik.com/Gaps2Ws.html] or downloaded and played as an AVI file [http://www.markgrechanik.com/Gaps2Ws.avi].

V. Implementation

In this section, we describe the low-level details of the system implementation, which we call *Gaps2Ws*, based on our approach. We explain how our choice of architecture is dictated by the real-world constraints of the enterprises, describe the components of the architecture, and delve into our implementation of the Designer. We conclude this section with the analysis of the code for web services output by the Designer.

A. The Architecture

The architecture of Gaps2Ws is shown in Figure 2. This choice of the architecture is influenced by the fact that in enterprise environments GAPs and web services are often located on different computers. Some GAPs are located on the same computer, but they may not be started at the same time due to certain constraints. For example, two instances of the same application cannot bind their sockets to the same port on the same computer, and subsequently, they cannot

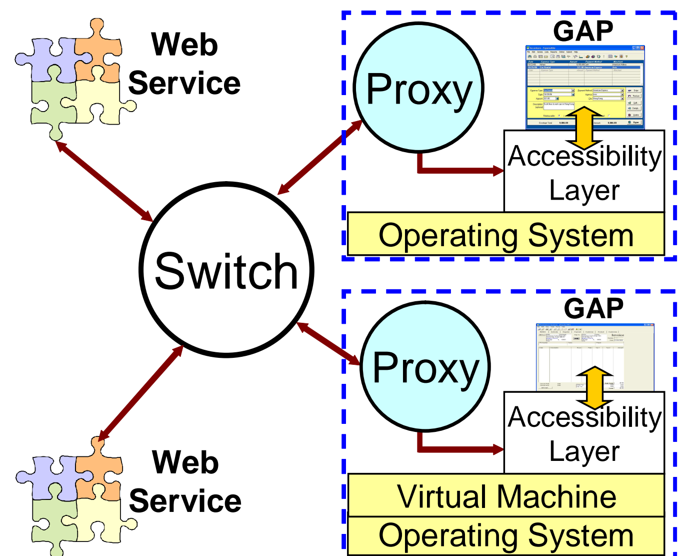

Fig. 2. The architecture of Gaps2Ws.

run simultaneously. This and some other conditions force administrators to distribute GAPs and web services across computers in enterprise environments.

Accessibility technologies cannot control distributed applications. If a web service and the GAPs it controls are located on different computers, then the service cannot use any accessibility technology to control these GAPs. A solution to this problem is to use proxies to control GAPs by sending commands to them from web services. A Proxy is a generic program that receives requests from web services, extracts data from GAPs in response to these requests, and sends the extracted data back to these web services. Proxies use the accessibility layer to control and manipulate GAPs.

The Switch is the central point for coordinating proxies in the distributed environment. It is a daemon program that collects information from proxies and makes decisions to which proxy to forward requests from web services. For example, if copies of the same GAP are installed on different computers, and web services need to control this GAP in response to requests from their respective clients, then the Switch assign the instances of this GAP to these services thereby enabling their execution in parallel.

The Switch is important for the Gaps2Ws architecture because it acts as an intermediary that enables web services and GAPs to run on different computers while presenting a common view to client programs. The clients can use web services transparently without knowing what computers the controlled GAPs are running on.

Since web services and GAPs may be moved around the enterprise computers for different reasons (e.g., to improve business processes or the performance of applications), the Switch provides the migration and location transparencies for web services and GAPs. Proxies register with the Switch under unique names, collect information about GAPs located on their computers, and send this information to the Switch. The Switch receives tables of GAPs from proxies on a regular basis, and it uses this information to direct requests from web services to appropriate GAPs.

When a method of a web service is invoked for the first time, it connects to the Switch and sends a registration request. From this request the Switch determines what GAPs are required to run the web service. The Switch looks up the GAP tables received from connected proxies, and once it finds the required GAPs, it sends requests to the corresponding proxies to reserve these GAPs for the web service. If no GAP is available, for whatever reasons, then the Switch puts this request on its internal queue and informs the web service that necessary GAPs are not currently available. The web service may either wait for the GAPs to become available or cancel the request.

When a client calls a method of the web service, the method sends request messages in a predefined format to the Switch. Each message carries information about the GAP, its elements, the actions on them that the web service needs to perform, and the information should be returned to the web service. The Switch forwards these messages to the appropriate proxies which in turn perform actions against the GAPs and set and retrieve values from GUI elements.

B. The Structure of GAPs

In event-based windowing systems (e.g., Windows), each GAP has a main window (which may be invisible), which is associated with the event processing loop. Closing this window causes the application to exit by sending the `DestroyWindow` event to the loop. The main window contains other GUI elements of the GAP. A GAP can be represented as a tree, where nodes are GAP GUI elements and edges specify that children elements are contained inside their parents. The root of the tree is the main window, the nodes are container elements, and the leaves of the tree are basic elements.

Each GUI element is assigned a category (class) that describes its functionality. In Windows, a basic class of all GUI elements is the class `window`. Some GUI elements serve as containers for other elements, for example, dialog windows, while basic elements (e.g., buttons and edit boxes) cannot contain other elements and are designed to perform some basic functions. This hierarchical containment is reflected in the object-oriented design of the Designer where classes representing container windows have fields that are instances of the classes that represent contained GUI elements.

C. Representing GUIs Programmatically

Recall our idea that GUI elements are programming objects whose values can be set and retrieved and whose methods can be invoked. As such, GUI elements can be represented as classes residing inside web services. Since GUI screens consist of different GUI elements, these screens are also GUI elements (i.e., windows) and can also be represented as classes whose fields are instances of the classes representing these GUI elements. Finally, GAPs consist of different GUI screens, and GAPs can be represented as classes whose fields are instances of the classes representing constituent GUI screens. We view GAPs as state machines, and web services control GAPs by transitioning them to different states using programming objects that represent these GAPs.

To summarize, GAPs are state machines whose states are defined as collections of GUI elements, their properties (e.g.,

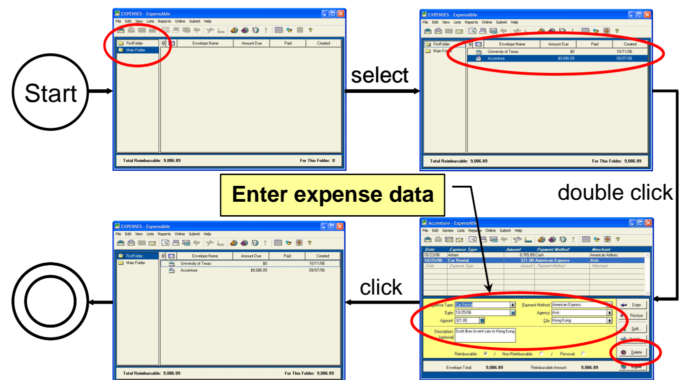

Fig. 3. The state machine describing transitions between screens of `Quicken Expensable 98`.

style, read-only status, etc.), and their values. When users perform actions they change the state of the GAP. In a new state, GUI elements may remain the same, but their values and some of their properties change.

An example of modeling transitions of `Quicken Expensable 98` as a state machine when entering expenses is shown in Figure 3. The states of the GAP are depicted using the screen images, and the transitions are shown with arrows labelled with actions that users perform on GUI elements. The elements against which these actions are performed are surrounded with ovals.

We distinguish between intermediate and final states. Consider clicking on a link in a web-based application. After loading a page that displays a progress GUI element, the browser is redirected to the destination page. Clearly, the page with the progress GUI is an intermediate state of the GAP, and the users are interested in the destination page, which is a final state. Users distinguish intermediate states from final states by visually inspecting GUI screens to check to see if required GUI elements are shown. In order to perform this function automatically, a method of a web service should inspect GAPs to analyze their structures and their GUI elements to detect intermediate and final states. This analysis is done by traversing GUI trees and comparing them to the trees that are recorded by the Designer when users perform operations on GAPs as described in Section IV-C.

D. The Designer Implementation

The Designer is the central component of the Gaps2Ws since it enables users to create web services from GAPs. Its implementation is based on representing GAPs as state machines.

The Designer takes inputs describing the states of the GAPs and generates classes whose methods control GAPs by setting and getting values of their GUI elements and causing actions that enable GAPs to switch to different states. When the user switches the GAP to some state, the Designer records this state by traversing the GUI tree of the GAP post-order using the accessibility technology. For each node of the tree (i.e., a GUI element), the Designer emits code for classes are linked to these GUI elements, and these classes contain methods for setting and getting values and performing actions on these elements. The Designer also emits the code that handles exceptions that may be thrown when web services control GAPs.

VI. Prototype Implementation

We implemented Gaps2Ws in Windows using C++ and Java. The prototype implementation is based on the MSAA toolkit version 2.0 and an MS XML parser, as all communications between the components of Gaps2Ws are in XML format. Our prototype implementation included the Designer, the Switch, and the Proxy. We wrote the Designer in Java and the rest of components of Gaps2Ws in C++. We used sockets as an interprocess communication mechanism. Our implementation contains close to `12,800` lines of code.

For our prototype we used Apache Axis 2, which is a platform for development and deployment of web services [http://ws.apache.org/axis2]. Under Axis, web services are written as Java classes and deployed by changing the `java` extension of files containing Java classes to the `jws` extension. Then these files are copied to a predefined directory under the Axis installation, and the web service is ready for use. This is the simplest way to deploy a web service using Axis. For a full-scale deployment of web services using the Web Services Deployment Descriptor under Axis we refer the reader to the Axis documentation [2]. In general, Gaps2Ws is not limited to any particular web services deployment platform.

VII. Experimental Evaluation

In this section we describe the methodology and provide the results of experimental evaluation of our approach (Gaps2Ws). We describe case studies in which we successfully create web services from two commercial closed and monolithic GAPs; we present various measurements collected during case studies and analyze them; and we conduct experiments to analyze the performance penalty when using GUIs to access GAP services versus invoking methods of pure programmatic web services. We show how to use our approach so that performance penalty incurred by communicating with GAPs through their GUI elements stays below some acceptable limit.

A. Case Studies

To demonstate our approach, we created web services from two commercial GAPs: `Quicken Expensable 98` (QE) and `ProVenture Invoices and Estimates` (PIE). QE and PIE are closed and monolithic GAPs that run on Windows. PIE allows users to create and print invoices, estimates, and statements, and to track customer payments and unpaid invoices. We discussed QE in Section IV-C.

Our experience confirms the benefits of our approach. We created web services for both QE and PIE without writing any additional code, modifying the applications, or accessing their proprietary data stores. Since QE and PIE are closed and monolithic commercial applications that neither expose any programming interfaces nor publish their data in known formats, we created web services from these GAPs using basic information about how users interact with them to accomplish tasks. We carried out experiments using Windows XP Pro that

GAP Name, State No	Number of GUI Elements			State XML, Bytes	Time, Sec
	Visible	Invisible	Used		
QE, State 1	152	23	16	46,345	0.3
QE, State 2	152	23	16	47,822	0.3
QE, State 3	193	36	28	73,339	0.4
QE, State 4	193	36	28	75,204	0.4
QE, State 5	152	23	12	46,838	0.3
PIE, State 1	226	51	35	96,018	0.5
PIE, State 2	207	46	41	106,225	0.5

TABLE I
Experimental results for using Gaps2Ws on GAPs QE and PIE.

ran on a computer with Intel Pentium IV 3.2GHz CPU and 2GB of RAM.

The results of this experiment are shown in Table I. The first column shows the name of the GAP and its comma-separated state identifier that is the sequence number of the screen. The next three columns show the number of visible and invisible GUI elements and the number of GUI elements including their parents in the hierarchy that are used as parameters or action targets in web services. The fifth column shows the size of the XML generated by the Designer to describe the given state of the GAP. Finally, the last column shows the time taken to generate and parse the state XML.

Even though XML is not inherent in Gaps2Ws, there is a possible problem with the size of the XML data especially considering that web services and GAPs that they control may be located on different computers. Sending large amounts of XML data from services to proxies may saturate the network traffic. A way to address this problem is to use XML data compression, or optimize the system to reduce the amount of data sent over the network. This is a subject of our future work.

In our case study, we compared the effort required to create web services using Gaps2Ws with the programming effort to create the same service by using the source code of GAPs. We created an application similar in functionality to QE. It took us approximately nine hours to create and test the QE's imitation (QEI) GAP. Then, we created a web service using Microsoft Visual Studio 2005 which has the state-of-the-art support for automating tasks for building web services. It took us approximately one hour to extract the code from the QEI, move it to the web service project, and compile and debug it using the Studio. Compared to that, it took us less than ten minutes to generate a web service using our approach.

B. Performance Considerations

Calling methods of programmatic web services is more efficient than invoking services of GAPs through their GUI elements. The additional overhead cost, `OC`, consists of the GAP startup time, the initialization time for the internal structures representing GUI elements, screen switching time, and communicating time between Proxies and GAPs. Common delay, `CD`, for both programmatic and GAP-based web services consists of network latency time of transmitting method call requests from clients to web services and delivering results back and the method execution time. The `GUI computation overhead (GCO)` ratio in percent, $GCO=\frac{OC}{CD} \cdot 100$, shows what percentage of the execution time is dedicated to handling GAPs and their GUI elements.

C. Performance Evaluation

The goal of the performance experiment is to evaluate how much performance penalty GAP-based web services incur versus pure programmatic ones. We designed the performance test to measure the reliability and sustainability of transaction processing throughput of our implementations of the QEI application. The test script simulated users logging in and then proceeding to individually enter `100` expenses. For each expense entered, the process was completed, with the last step in this process being the return to the initial screen of the GAP, followed by a logout at the end of the script. Each virtual user therefore completes `100` individual transactions during a user session. The test was run at a user load for duration of `24` hours.

We repeated this test four times for our implementations of QEI to collect more data. The first test was run with the QEI implemented as a purely programmatic web service with no GAPs used. The second test was run against the GAP QEI whose screens were reused by returning to original screens, not restarting the GAP. The third test was run against the GAP QEI which was restarted after each transaction. The last test was run with QEI whose GUI contained different multimedia elements (e.g., animations and bitmaps). We report an average time per transaction for each test.

Experimental results from evaluating how much performance penalty these GAP-based web services incur versus pure programmatic ones are shown in Figure 4. The vertical axis shows the average time in seconds per transaction, and the bars correspond to the tests. The fastest transaction takes on average `1.6` seconds when no GAPs are used for the QEI web service, that is the service is purely programmatic. The performance drops when the GAP QEI is used to encapsulate the functionality required by the web service. The average time per transaction increases to `1.9` from `1.6`, which is 18.8% increase. The difference between these average transaction times is 0.3 second, which we attribute to the overhead of the GUI computations.

The situation worsens for the third test when the GAP is required to restart every time the web service is run. The overhead associated with restarting of the GAP increases the average time per transaction to `3.2` seconds. Finally, when the GAP uses multimedia images and animations, the performance becomes worse, taking on the average time per transaction `6.3` seconds.

D. Recommendations

Choosing Gaps2Ws versus writing programmatic web services is a matter of trade-offs between the development effort and the resulting performance. If the performance of the web

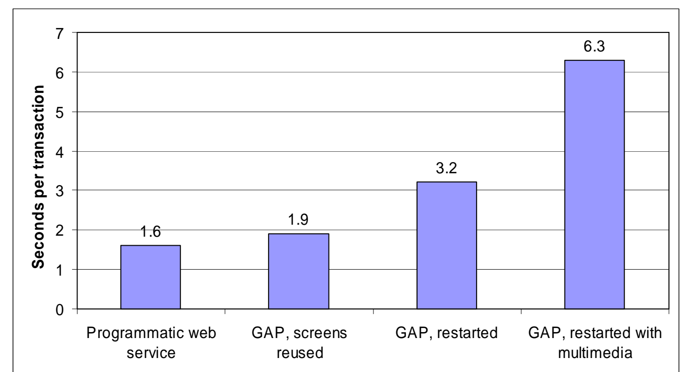

Fig. 4. Transaction throughput for web services.

service is a critical issue, then developing a programmatic web service is the right choice. However, when minimizing cost is important while performance should be acceptable, using Gaps2Ws allows users to achieve these objectives.

Recall that the GUI overhead `GCO` is inversely proportional to the common delays `CD`, such as network latency and GAP backend computations. At one extreme, `CD` is much smaller than the `OC`, and the `GCO` is high. For example, if the `OC` is one second and the `CD` is one tenth of a second, then the `GCO` is `1,000`%. Since the `GCO` is high, users should consider to develop a programmatic web service. At the other extreme, `CD` is much higher than the `OC`, and the `GCO` is small. For example, if the `OC` is one second and the `CD` is `20` seconds, then the `GCO` is `5`%.

Based on our conversations, many professionals who build, deploy, and maintain web services are willing to consider up to `10`% of performance penalty if they can reduce the development effort. We observed that for many commercial applications backend computations take from five to fifteen seconds. It means that for the `GCO` to be less than `10`%, its absolute value should be between `0.5` to `1.5` seconds, which is consistent with the `GCO` of `0.3` second which we measured in our performance experiment with the QEI.

Since GAPs consume significant CPU time for GUI painting when images and animations are included, using these applications for web services may not be possible for performance reasons. In practice, clients use web services via the Internet, and web services use GAPs via the LAN. From this perspective the performance penalty incurred by using GAPs is minimal since the low-level communication mechanisms such as transmission, marshaling and unmarshaling network data have the largest overhead common to all solutions.

E. Limitations

In general, Gaps2Ws may not work well with GAPs whose GUIs are dynamically created. However, the number of such GAPs is small, and most GUIs are stable and may have small changes between releases, which happen infrequently. It is a bigger problem with web-based applications whose GUIs change relatively frequently. In future we may consider machine learning approaches that will allow Gaps2Ws to learn the GUI layout and to locate required elements automatically as GUIs change dynamically.

When attempting to run two web services in parallel, we found that multiple instances of QE are prevented by design from running on the same computer. This problem can be solved by installing a copy of QE on a different computer and configuring its Proxy and the Switch.

In general, managing many instances of the same GAP is difficult. For example, when running web-based applications, they open many popup windows. These windows and processes that control them are not linked explicitly to the application that opened them. Thus, when two or more web-based applications ran on the same computer simultaneously, data from these applications may be mixed. We are currently working on solving this problem.

VIII. Conclusion

We proposed a novel generic approach for creating web services from GAPs. This approach combines a nonstandard use of accessibility technologies for accessing and controlling GAPs in a uniform way with a visualization mechanism that enables nonprogrammers to create web services by performing point-and-click, drag-and-drop operations.

We built a tool based on our approach, and we used this tool to create web services that control two closed and monolithic commercial GAPs. We determined that our approach is especially effective when back-end computations performed by GAPs exceed three seconds thereby reducing the GUI computational overhead, which is shown in our experiments to be approximately `0.3` second, to less than `10`%. Our evaluation suggests that our approach is effective and it can be used to create web services from nontrivial legacy GAPs.

We believe that our approach has enormous potential. It could be used for data integration, GAP reuse, collaborative computing, and application migration to new platforms. Its key advantage is that all of these cases involve minimal development efforts. It offers, for example, an attractive alternative to the way web services are created from legacy applications.

References

[1] Building software that is interoperable by design. *http://www.microsoft.com/mscorp/execmail/2005/02-03interoperability-print.asp*.

[2] Documentation for Apache Axis 1.2. *http://ws.apache.org/axis/java/index.html*.

[3] Screen-scraping entry in Wikipedia. *http://en.wikipedia.org/wiki/Screen_scraping*.

[4] Section 508 of the Rehabilitation Act. *http://www.access-board.gov/508.htm*.

[5] *IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries.* Institute of Electrical and Electronics Engineers, January 1991.

[6] B. Buck and J. K. Hollingsworth. An API for runtime code patching. *Int. J. High Perform. Comput. Appl.*, 14(4):317–329, 2000.

[7] C. Ferris and J. A. Farrell. What are web services? *Commun. ACM*, 46(6):31, 2003.

[8] M. Grechanik, D. S. Batory, and D. E. Perry. Integrating and reusing GUI-driven applications. In *ICSR*, pages 1–16, 2002.

[9] J. R. Larus and E. Schnarr. EEL: Machine-independent executable editing. In *PLDI*, pages 291–300, 1995.

[10] M. Li, O. F. Rana, M. S. Shields, and D. W. Walker. A wrapper generator for wrapping high performance legacy codes as Java/CORBA components. In *SC*, 2000.

[11] R. C. Miller. End-user programming for web users. In *End User Development Workshop, Conference on Human Factors in Computer Systems*, 2003.

[12] R. C. Miller and B. A. Myers. Integrating a command shell into a web browser. In *USENIX Annual Technical Conference, General Track*, pages 171–182, 2000.

[13] B. A. Myers. User interface software technology. *ACM Comput. Surv.*, 28(1):189–191, 1996.

[14] E. Tilevich, Y. Smaragdakis, and M. Handte. Appletizing: Running legacy java code remotely from a web browser. In *ICSM*, pages 91–100, 2005.

[15] E. Wohlstadter, S. Jackson, and P. T. Devanbu. Generating wrappers for command line programs: The cal-aggie wrap-o-matic project. In *ICSE*, pages 243–252, 2001.