Composing Integrated Systems Using GUI-Based Applications And Web Services

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

Abstract— **Integrated systems are composed of components that exchange information (i.e., interoperate [5]). These components include *Graphical User Interface (GUI) APplications (GAPs)* and web services. It is difficult to make GAPs interoperate, especially if they are closed and monolithic. Unlike GAPs, web services are applications that are designed to interoperate over the Internet.**

We propose a novel generic approach for creating integrated systems by composing GAPs with each other and web services efficiently and non-invasively. 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 compose integrated systems by performing point-and-click, drag-and-drop operations on GAPs and web services. We built a tool based on our approach, and using this tool we created an integrated application that controls two closed and monolithic commercial GAPs and third-party web services. Our evaluation suggests that our approach is effective, and it can be used to create nontrivial integrated systems by composing GAPs with each other and web services.

I. Introduction

Integrated applications consist of components that exchange information (i.e., to interoperate [5]). Building integrated applications is difficult and expensive, since in addition to building components of these applications programmers should define protocols and implement the functionality for data exchanges between these components.

Components of integrated applications include *Graphical User Interface (GUI) APplications (GAPs)* and web services. Organizations use legacy GAPs to assist business operations. 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. Thus, while it is desireable to use GAPs as components in integrated applications, it is often difficult to add functionality to GAPs to enable them to interoperate with other applications.

Web services are software components that interoperate over the Internet, and they have gained widespread acceptance partly due to the business demand for applications to exchange information [9]. Unlike GAPs, using web services enables organizations to quickly build integrated systems by composing these services for information exchange. By composing legacy GAPs with each other and web services into integrated systems, organizations can support their business processes better with these systems [8].

Integrating legacy GAPs with each other and web services is important for most organizations that have invested heavily in a variety of GAPs from multiple vendors [1]. Changing source code of GAPs to make them interoperable is difficult because of brittle legacy architectures, poor documentation, significant programming effort, and subsequently, the large cost of these projects. Modifying third-party GAPs may not even be possible since organizations often do not have access to the source code. Given the complexity of GAPs and the cost of making them interoperable, a fundamental problem of interoperability is how to build integrated systems by composing GAPs with each other and web services efficiently and non-invasively.

Our main contribution is a novel generic approach for composing GAPs with each other and web services into integrated systems. This approach combines a nonstandard use of accessibility technologies for composing GAPs written in different languages and running on different platforms in a uniform way with a visualization mechanism that enables nonprogrammers to compose integrated systems by performing point-and-click, drag-and-drop operations against closed and monolithic GAPs and web services. 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 composing GAPs with each other.

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

II. A Motivating Example

E-procurement systems (EPS) are critical since they can influence all areas of an organization's performance [7]. Businesses employ elaborate EPSes that often consist of different GAPs assisting different steps of the purchasing process. In EPSes, the *rule of separation of duty* requires that operations be separated into different steps that must be done by independent persons (agents) in order to maintain integrity. With the separation of duty rule in place, no person can cause a problem that will go unnoticed, since a person who creates or certifies a transaction may not execute it. Implementing this rule results

in agents using different GAPs that provide different services for different steps of the purchasing process.

Consider a typical e-procurement scenario. Employees order items using an electronic shopping cart service of the web-based application *OfficeDepot (OD)*. Department managers review selected items in the shopping cart, approve and order them, and enter the ordered items into *Quicken Expensable 98 (QE)*, a third-party closed and monolithic Windows GAP that the company uses internally to keep track of purchases.

The OD service sends a notification to a company accountant, who uses a closed and monolithic GAP called *ProVenture Invoices and Estimates (PIE)* to create invoices for ordered goods. When the ordered goods are received from OD, a receiving agent compares them with the entries in QE. The accountant can view but cannot modify records in QE, and likewise, no other agent but the accountant can insert and modify data in PIE. If the received goods correspond to the records in QE, the receiving agent marks the entries for the received goods in QE and notifies the accountant. After comparing the invoices in PIE with the marked entries in QE and determining that they match, the accountant authorizes payments.

In this example, each procurement agent uses different GAPs to accomplish different goals. Sometimes, several GAPs can be used to accomplish a single goal, and agents have to transfer data between these GAPs and perform other operations manually. Clearly, automating these activities is important in order to improve the quality and the efficiency of business services.

Consolidating disparate components into integrated EPSes enables enterprises to achieve a high degree of automation of their purchasing processes [8]. One supporting function of an integrated system is to extract information about ordered items from the service OD and create and enter invoices into PIE using this information. Once the payments are processed, the user marks invoices in PIE as paid, and the information about these invoices should be extracted from PIE and entered as expenses into QE. There are many other functions of the integrated EPS that involve interoperating GAPs with each other and web services.

An important function of an integrated EPS is to extend the functionalities of GAPs with web services. It means that users can continue to work with legacy GAPs, however, certain new functions will be delegated to web services. For example, when users modify expenses using QE, they may be required to submit the information about modified expenses to a web service that verifies these expenses using some business rules. These and many other functions reflect the need to compose GAPs with each other and web services in integrated systems.

III. How Our Approach Works

We present a birds-eye view of how our approach works by giving examples of composing GAPs with each other and web services to create integrated systems. First, we describe how to compose a simple integrated EPS from QE, PIE, and a web service, and then we show how to extend the functionality of QE with a web service.

A. Composing GAPs with Each Other And a Web Service

We have built a tool called *COmposer of INtegrated Systems (Coins)* that enables users to create intergrated systems by composing GAPs with each other and web services. Coins outputs integrated systems as composite web services that control and manipulate GAPs and other web services thereby making these components interoperate with one another. Specifically, we show how to compose an integrated system that extracts information about ordered items from the service OD and creates and enters invoices into PIE using this information. After these invoices are paid, the information about them is extracted from PIE and entered as expenses into QE.

The front end of Coins is shown in Figure 1. Using Coins, a user enters the name of the composite web service and the name of the exported method of this service (the defaults are `Service1` and `operation1`) respectively. The user also specifies that the service controls GAPs QE and PIE and the web service OD by providing information about them (i.e., their locations and *Web Service Definition Language (WSDL)* data). This information is shown in the leftmost tab (i.e., `Service Explorer`) of Coins.

Next, the user enter an expense and an invoice into the GAPs QE and PIE correspondingly. Specifically, the user selects an expense envelope on the first screen of QE, and then double clicks on the entry in the envelope list box. These actions cause QE to switch to the expense entry screen. This and other screens of the GAP are captured and shown in the rightmost tab (i.e., `Screen View`) of Coins.

The purpose of interacting with GAPs is to allow Coins to record the structures of the screens and user actions on the GAPs and then transcode these actions into programming instructions that the resulting composite web service will execute. To do that, Coins intercepts selected user-level events using the *accessibility layer*[1]. These events allow Coins to record the sequence of screens that the user goes through as well as the actions that the user performs on GUI elements.

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

At the design time, the user should specify how to exchange data between components. Specifically, the user determines what GUI elements will receive the values of the properties of the web service OD and the values of GUI elements of other GAPs. For example, the value of the property `Item` of the service OD should be put into the GUI element labeled `Description` of the invoice screen of PIE, which in turn should be put into the GUI element labeled `(optional)` of the expense screen of QE.

[1]We discuss accessibility technologies in Section IV-A.

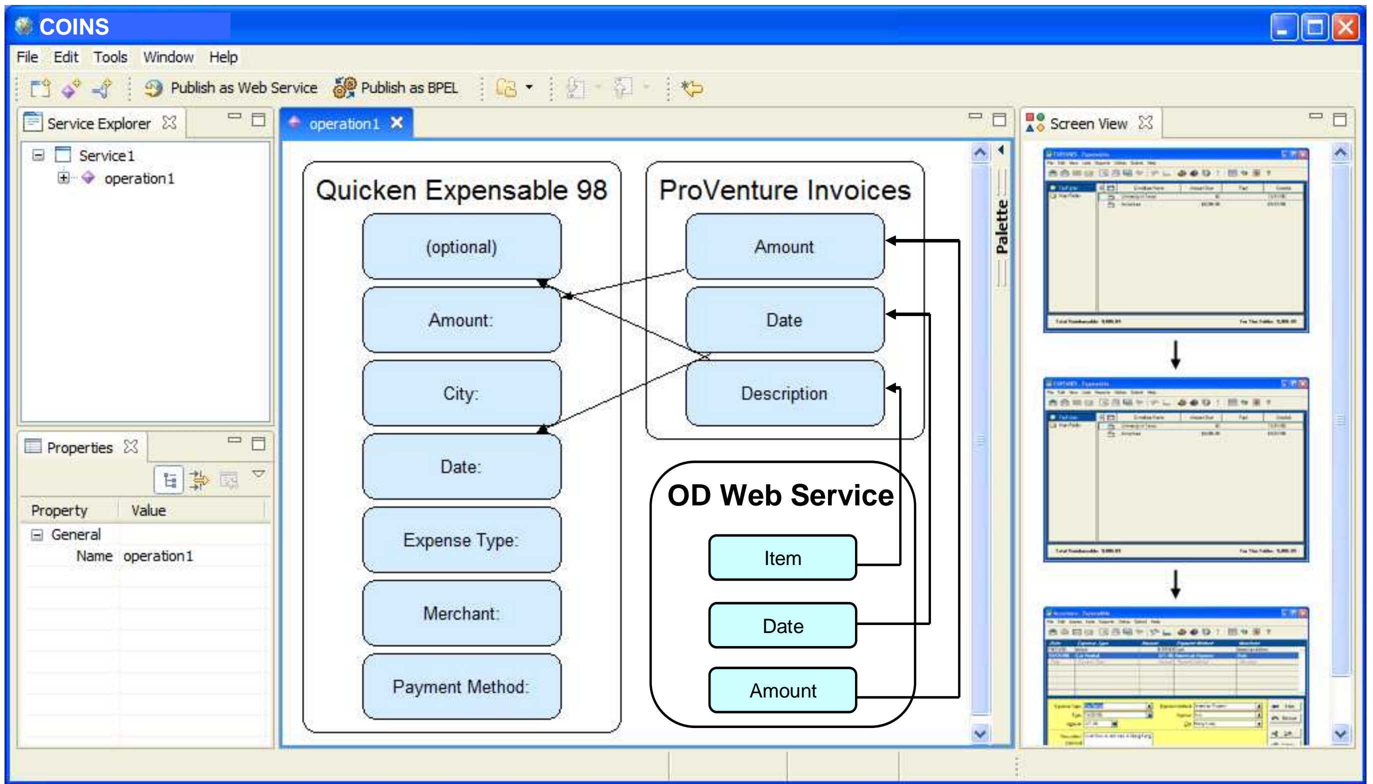

Fig. 1. The front-end of Coins.

To create mappings between these properties and GUI elements, the user moves the cursor over some GUI element of a GAP, and Coins uses the accessibility *Application Programming Interface (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 Coins' middle tab labeled with the name of the exported method (i.e., `operation1`) of the composite web service. After releasing the mouse button, the dragged element is dropped onto the Coins' dataflow palette under the label of the corresponding component.

Once the user has dragged-and-dropped all required GUI elements and loaded the description of web services from their WSDL files, it is time to connect these elements and properties of the loaded services with arrows that specify the directions of data exchanges. For example, by drawing an arrow between the element `Amount` of the PIE and the element `(Amount)` of QE, the user specifies that the data from the corresponding GUI element of PIE will be transfered to the GUI element of QE. While it is possible to specify how to transform the data during exchange, we do not consider these modifications in this paper for simplicity.

In addition, the user specifies what action(s) should be performed on GUI elements and what methods of the web services to call to initiate data exchange. For example, clicking on the tab `Invoice` in PIE initiates the procedure for extracting invoices. These actions are recorded as part of the workflow which can be exported as a *Business Process Execution Language (BPEL)* program.

The resulting composite service is published by clicking on the button `Publish as Web Service` on the Coins' toolbar. Coins uses the information captured for each screen and input elements to generate Java code for the composite web service and deploy it to a web services platform such as Apache Axis. When this service is called from a client, the method `operation1` uses the accessibility interfaces to control and manipulate the GAPs and the web service to exchange information. A short movie demonstrating how Coins works is available at our website [www.markgrechanik.com]. It can be viewed inside a browser [http://www.markgrechanik.com/Coins.html] or downloaded and played as an AVI file [http://www.markgrechanik.com/Coins.avi].

B. Extending GAPs With Web Services

Legacy GAPs may be required to support new business processes. For example, a new business procedure may require that users submit the information about entered and modified expenses to a web service that verifies these expenses using some business rules before saving these expenses in QE. Since QE has been used for many years, integrating it with the new service allows the business to achieve new functionality at a low cost.

Coins allows users to extend the functionality of GAPs by integrating them with web services. The user connects GUI elements of QE with properties of the web service in the middle pane of Coins with arrows from the palette toolbox

thereby specifying how data is transferred from the GAP to the service. Then, the user selects a method of the web service and determines how to invoke it. This method is invoked when a user performs some action on a GUI element (e.g., clicks a button). The values of GUI elements are passed as parameters to the invoked method, or they are used to set properties of the web service before the method is invoked.

In addition, the user specifies how to use the return values of the invoked method. They can be put in the selected GUI elements of the GAPs, or they can be displayed in message dialogs. The user can select an action in response to certain return values of the invoked method. Examples of these actions are terminating the GAP or going to the previous screen of the GAP whenever the latter action is possible.

IV. Elements of Our Solution

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-A.

A. 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 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.

B. Hooks

Hooks are user-defined libraries that contain *callback functions* (or simply *callbacks*), which are written in accordance with certain rules dictated by accessibility technologies. Hooks are important for Coins because they enable users to extend the functionality of GAPs, specifically to integrate them with web services without changing GAPs' source code. Writing hooks does not require any knowledge about the source code of GAPs.

In our approach, a hook library is generic for all GAPs, and its goal is to listen to events generated by the GAP into which this hook is injected as well as to execute instructions received from integrated systems. An example of an instruction is to disable a button until certain event occurs. The power of hook libraries is in changing the functionalities of existing GAPs without modifying their source code.

Main functions of the generic hook are to receive commands to perform actions on GUI elements, to report events that occur within GAPs, and to invoke predefined functions in response to certain commands and events. Since accessibility layers are supported by their respective vendors and hooks are technical instruments which are parts of accessiblity layers, using hooks is legitimate and accepted to control and manipulate GAPs. In addition, writing and using hooks is easy since programmers use high-level accessibility API calls, and they do not have to deal with the complexity of low-level binary rewriting techniques.

When a target GAP is started, the accessibility layer loads predefined hook libraries in the process space of this applications and registers addresses of callbacks that should be invoked in response to specified events. Since hooks "live" in the process spaces of GAPs, their callbacks can affect every aspect of execution of these GAPs.

V. Architecture

The architecture of Coins is shown in Figure 2. Solid arrows show command and data flows between components, and numbers in circles indicate the sequence of operations in the workflow.

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

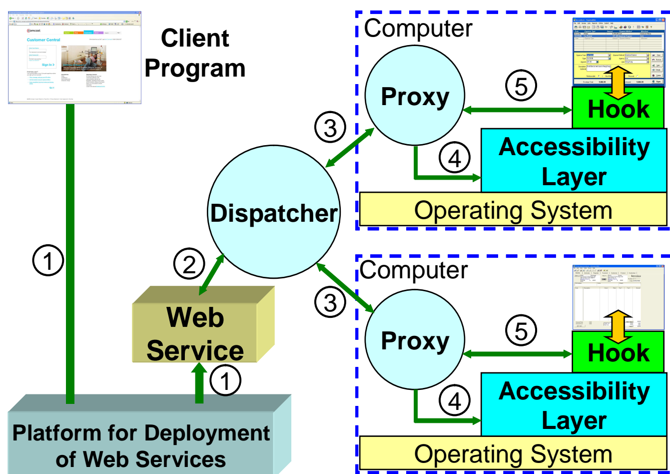

Fig. 2. The architecture of Coins.

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 an integrated system (e.g., a composite 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 integrated systems. A Proxy is a generic program that receives requests from integrated systems, extracts data from GAPs in response to these requests, and sends the extracted data back to these integrated systems. Proxies use the accessibility layer to control and manipulate GAPs.

The Dispatcher 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 composite web services that represent integrated systems. For example, if copies of the same GAP are installed on different computers, the Dispatcher assign the instances of this GAP to different requesting clients thereby enabling their execution in parallel.

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 Dispatcher provides migration and location transparency for web services and GAPs. Proxies register with the Dispatcher under unique names, collect information about GAPs located on their computers, and send this information to the Dispatcher. The Dispatcher 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 the composite web service, which represents the integrated system is invoked for the first time by its client `(1)`, the service connects to the Dispatcher and sends a registration request `(2)`. From this request the Dispatcher determines what GAPs are required to run the web service. The Dispatcher 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 `(3)`.

After the Proxy starts the GAP, it uses the accessibility layer to inject the hook library into the GAP `(4)`. The hook spawns a thread within the GAP's process in order to establish a communication channel with the corresponding Proxy. Using this channel, the Proxy can send commands and receive notifications of the events that occur within the GAP and for which callbacks from the hook are registered `(5)`. Thus the generic hook can be viewed as a virtual machine that can be used to control and manipulate GAPs.

VI. Code Generation

Coins generates code for the resulting integrated system (i.e., a composite web service). The structure of the code reflects dependencies between GUI elements in 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 terminating the loop. The main window contains other GUI elements of the GAP. A GAP can be represented as a tree, where nodes are GUI elements and edges specify children elements that 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 (e.g., buttons or edit boxes).

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

Coins 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, Coins 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), Coins 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. Coins also emits the code that handles exceptions that may be thrown when web services control GAPs.

VII. Prototype Implementation

We implemented Coins 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 Coins are in XML format. Our prototype implementation included the front end, the Dispatcher, the composite service code generator, the Proxy, and the hook. We wrote the front end and the generator in Java and the rest of components of Coins in C++. We used sockets as an

interprocess communication mechanism. For our prototype we used Apache Axis 2, which is a platform for development and deployment of web services [http://ws.apache.org/axis2]. 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]. Our implementation contains close to `12,800` lines of code.

VIII. Experimental Evaluation

In this section we describe the methodology and provide the results of experimental evaluation of Coins. We describe case studies in which we successfully integrate two commercial closed and monolithic GAPs and a web service in an instance of the EPS which is described in Section II; we demonstrate how batch data exchanges result in significant performance gain using our approach; and we conduct experiments to analyze the performance penalty when using GUIs to access GAP services in integrated systems 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 Study

To demonstate our approach, we created an E-procurement system (EPS) from QE and PIE, as we described in Section II. Recall that QE and PIE are closed and monolithic commercial GAPs that run on Windows. QE allows users to enter and track expenses, and PIE allows users to create and print invoices, estimates, and statements, and to track customer payments and unpaid invoices.

Our experience confirms the benefits of our approach. We created an integrated EPS from both QE and PIE and a web service without writing any additional code, modifying the applications, or accessing their proprietary data stores (see Section III). We carried out experiments using Windows XP Pro that ran on a computer with Intel Pentium IV 3.2GHz CPU and 2GB of RAM.

In our case study, we compared the effort required to create an integrated EPS using Coins with the programming effort to create the same service by using the source code of GAPs. We created applications similar in functionalities to QE and PIE respectively. It took us approximately fourteen hours to create and test our imitations of QE and PIE which we call IQE and IPIE respectively. Then, we created an integrated EPS. It took us approximately three and half hours to extract the code from the IQE and IPIE, move it to the integrated system project, and compile and debug it. Compared to that, it took us less than fifteen minutes to generate an integrated system using Coins.

B. Performance Considerations

Calling methods directly 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 integrated systems consists of network latency time of transmitting method call requests between components and delivering results back and the method execution time. The `GUI computation overhead (GCO)` ratio in percent, $\mathtt{GCO}=\frac{\mathtt{OC}}{\mathtt{CD}} \cdot 100$, 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 integrated incur versus pure programmatic ones. In addition, we show that using batch data exchanges results in significant performance gain.

1) Performance Stress Test: We designed the performance test to measure the reliability and sustainability of transaction processing throughput of our implementation of the EPS system. The test script simulated users who individually order `100` items from the web service OD. For each invoice entered in IPIE, the process included extracting this invoice and creating a corresponding expense in IQE, with the last step in this process being either the return to the initial screen of the GAP, or the termination of the GAPs. 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 in order to minimize various effects of other applications and services running on the same computer as well as network irregularities over the extended period of time.

We repeated this test four times for our implementations of IQE and IPIE to collect more data. The first test was run with the IQE and IPIE implemented as purely programmatic components (libraries) with no GUI interfaces used to transfer data. The second test was run against IQE and IPIE whose screens were reused by returning to original screens, not restarting the GAPs. The third test was run against the IQE and IPIE which were restarted after each transaction. The last test was run with IQE and IPIE whose GUIs 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 data transfers incur versus pure programmatic ones are shown in Figure 3. The vertical axis shows the average time in seconds per data exchange transaction, and the bars correspond to the tests. The fastest transaction takes on average `0.8` seconds when no GAPs are used, that is the data is passed purely programmatically between IQE and IPIE libraries. The performance drops when GUI elements of IQE are used to encapsulate the functionality required to transfer the data. The average time per transaction increases to `2.2` seconds from `0.8` seconds, a 175% increase. The difference between these average transaction times is 1.4 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 data exchange is performed. The overhead associated with restarting of the GAP increases

the average time per transaction to 4.6 seconds. Finally, when the GAP uses multimedia images and animations, the performance becomes worse, taking on the average time per transaction 7.2 seconds.

2) Batch Data Exchanges: Coins enables users to specify batch data exchanges between components of integrated systems. Sending messages between components is delayed in batch message transfers until a batch message containing many smaller messages is formed and transmitted at once. The combined overhead associated with sending many small messages is higher than the overhead of sending one larger message that contains these small messages. Sending separate small messages involves adding control information in headers that specify additional information for message transfer, invoking functions that create and parse these messages, and often having to perform additional extraction operations on GUIs. Extracting all data from GUI elements and sending them to the destination in one single message enables users to amortize a one-time overhead over many data items in the message.

In addition, since the network latency time is one of the major contributors to the overhead, sending fewer messages may result in the reduction of this overhead. Specifically, sending one large message instead of many small messages reduces the communication overhead between the Dispatcher and Proxies. Also, when sending many messages, next message is usually sent when the receipt of the previous message is acknowledged by a special response message. Sending and waiting for these acknowledgement messages adds unnecessary computation and communication overhead, which can be significantly reduced using batch transfers.

The goal our experiment is to show that exchanging messages in batches yields better performance of the integrated system. The graph showing the dependency of transfer time per item from the number of items transferred between GAPs is shown in Figure 4. The horizontal axis shows the number of data items transferred per transaction. An average size of the data item is approximately 360 bytes. The vertical axis shows the time it takes to transfer an item from IPIE to IQE. For a single item it takes approximately 2.1 seconds to transfer a data item considering the GAP computational overhead.

As data items are transferred in batches, the amount of time it takes to transfer an item drops approximately 14 times to close to 0.15 second as the number of the items in a batch grows to 500. Correspondingly, the size of the XML message containing these data items grows from 1.5Kb to 130Kb. However, transferring 1,000 items instead of 500 reduces the transfer time per item by only 1.9 times. Our explanation is that it takes longer to create and parse large messages as well as to transfer them, and this additional overhead dwarfs the reduction of the transfer time per message. Still, the our experiment shows that it is possible to gain significant performance by performing batch transfers between GAPs.

D. Recommendations

Choosing Coins versus writing programmatic integrated systems is a matter of trade-offs between the development effort and the resulting performance. If the performance of the integrated system is a critical issue, then developing a programmatic application is the right choice. However, if minimizing cost is important, using Coins allows users to reduce development cost significantly while keeping performance overhead within acceptable limits.

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 developing a programmatic integrated system. 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 with many professionals who build, deploy, and maintain integrated systems, these professionals are willing to consider up to a 10% performance penalty if they can reduce development efforts. 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 1.4

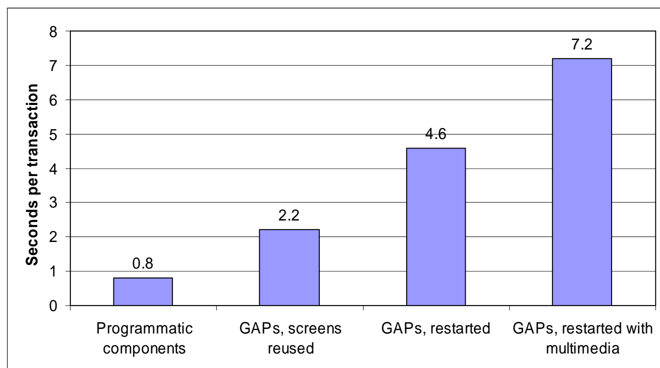

Fig. 3. Transaction throughput for web services.

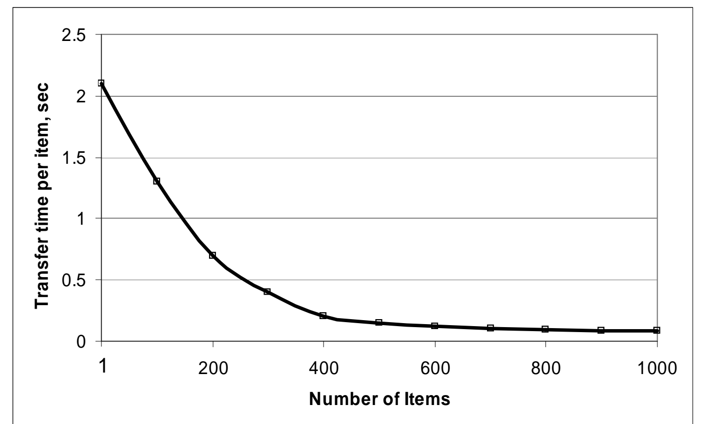

Fig. 4. Dependency of transfer time per item from the number of items transferred between GAPs using the batch exchange.

seconds which we measured in our performance experiment with the data transfer between IQE and IPIE.

Since GAPs consume significant CPU time for GUI painting when images and animations are included, using these applications for integrated systems may not be possible for performance reasons. In practice, clients use composite web services as integrated systems via the Internet, and these 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, Coins may not work well with GAPs whose GUIs are frequently modified since it would require users to regenerate integrated systems to adjust for new GUIs. 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 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.

IX. Related Work

UniFrame is a framework for building integrated systems by assembling pre-developed heterogeneous and distributed software components [15]. The glue/wrapper code that realizes the interoperation among the distributed and heterogeneous software components can be generated from the a descriptive model. Unlike UniFrame, Coins does not require users to write code for models, and it does not require the knowledge of the source code of components.

A web browser-shell integrates a command interpreter into the browser's location box to automate HTML interfaces [13]. 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.

Code patching [6] and binary rewriting [11] techniques modify the binary code of executable programs in order to control and manipulate them when integrating these programs into composite systems [10]. 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 [14] [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 [12]. Our approach differs fundamentally from other screen-scraping techniques since it modifies GAPs to extend their functionalities while, by definition screen scrapers only deliver information describing GUIs. In addition, Coins does not depend on parsing a scripting language that describes the GUI, and therefore it is more generic and uniform.

X. Conclusion

We proposed a novel generic approach for creating integrated systems by composing GAPs with each other and web services efficiently and non-invasively. 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 compose an integrated system from two closed and monolithic commercial GAPs and web services. Our case study showed that a key advantage of Coins is that it involves minimal development efforts. Our evaluation suggests that our approach is effective and it can be used to create integrated systems from nontrivial legacy GAPs and web services.

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] D. Burt, D. Dobler, and S. Starling. *World Class Supply Management: The Key to Supply Chain Management.* McGraw-Hill Irwin, July 2002.

[8] A. N. K. Chen and B. B. M. Shao. Web services enabled procurement in the extended enterprise: An architectural design and implementation. *J. Electron. Commerce Res.*, 4(4):140–155, 2003.

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

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

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

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

[13] 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.

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

[15] W. Zhao, B. R. Bryant, C. C. Burt, R. R. Raje, A. M. Olson, and M. Auguston. Automated glue/wrapper code generation in integration of distributed and heterogeneous software components. In *EDOC*, pages 275–285, 2004.