Form2WSDL Project Final Report

Final Year Project Final Report

Ross Shannon

A thesis submitted in part fulfilment of the degree of BSc. (Hons.) in Computer Science with the supervision of Dr. Nicholas Kushmerick and moderated by Dr. Mel Ó Cinnéide.

Department of Computer Science

University College Dublin

13 March 2005

Abstract

“Web Services” are a standard way of invoking procedures across the Internet. They allow programs running on any client to easily interface with and execute scripts on a server. For example, a script could be written to automatically buy a book from a bookseller’s website without anyone having to visit the site themselves. Though the possibilities are very exciting, the web services vision is being held back simply because not enough publicly-available web services have been written. This project was created to investigate the possibility of expanding the range of available services by converting ordinary human-orientated HTML forms that exist on many websites into equivalent web service descriptions, and allowing them to be invoked as normal web services.

Table of Contents

1 Introduction

A web service can be thought of as a method that can be invoked over the web from inside a client application. An application sends a request to a web service, which then processes the request and sends back a response. For instance, a programmer could ask a bookseller's web service what the current price of a particular book is by sending the service a book's ISBN number. All the programmer needs to know are the names of the available methods and what format to send their request in. To ascertain these details, the programmer can look at the web service's description.

Web services are described in a language called WSDL (Web Services Description Language), which is an application of XML (eXtensible Markup Language), and so is familiar in syntax. A WSDL file will have a listing of all of its methods, along with the format of data that it will accept and return [1]. The WSDL file (and optional embedded documentation) forms the necessary interface that allows the programmer to ascertain how to call on one of these remote methods to get data that can be manipulated by their own program.

Ideally, there would be a large catalogue of readily available services so that programmers can choose the ideal method to solve whatever problem they're facing, instead of having to write the code themselves. However, apart from some more common operations, this is not the case. A relatively small number of companies have created APIs (Application Programmer Interfaces) to allow programmers to communicate with their web services, among them Amazon and Google. These two existing services allow a range of opportunities that have already been put into practice[1], from text spell-checking to specialised searches. The programs that have resulted from these APIs include a browser modification that presents an interface that allows a user to search for and buy books from Amazon without ever needing to visit the company's website[2].

Today, there are an enormous amount of forms on websites that are used to query their databases and otherwise retrieve and process data. These, in effect, behave in the same way as a corresponding web service would, by taking input from the form and passing it to a backend script which will interpret this data and optionally pass back some output. The problem with this situation is that it requires a user to go to the website, figure out how to use the form and fill it in themselves, rather than allowing a program to easily send the required data directly to the script.

This project is concerned with using a program to convert these ubiquitous web forms into an arrangement in which they may be treated as web services. This report first describes the research done in preparation for this project. Following that is a discussion of the problem of parsing HTML forms for the data necessary to build up a WSDL model for each. Finally, there is a description of various aspects of the project's implementation as a Perl script running on a publicly-accessible server.

[1] The Google Web APIs can be found at http://www.google.com/apis/. For an example of the uses others have found for these services, consult the collection of tools built with the Google API listed at http://www.voelspriet.nl/googletools.htm.

[2] The Amazon Web APIs can be found at http://www.amazon.com/gp/aws/landing.html. One well known project that has made use of the API is The Mozilla Amazon Browser extension project: http://mab.mozdev.org/.

2 Background Research

2.1 Motivations

Though there are obvious real-world benefits from offering a web service, very few companies have taken the time to create one. This in part can be attributed to the difficulty of writing a WSDL description for a web service by hand, and the dearth of tools available to convert pre-existing human-consumable interfaces into machine-readable web services. Though a few directories of available services do exist[3], the number of ready-made services is still low in comparison to the number of websites with interfaces for accessing scripts and retrieving data.

Thus, the landscape at the moment is that there are a small number of web services and a very large number of web forms. Web services are currently entering a transitional period, as many developers are aware of their existence and would like to be able to use them in their own projects, but the data they require may not yet be retrievable from an existing web service. However, a developer may be able to get the required data manually by visiting a website and filling in a form.

The motivation for the Form2WSDL project is to bridge this gap in some way. If a program could be sent to fetch a page from the web, parse it for forms and then output an equivalent WSDL description of those forms for a developer to use in writing their own program, the available options to a programmer looking for useful web services to solve their particular problem will increase dramatically.

2.2 Anatomy of a Web Service

The WSDL file that describes the methods available from a web service, and the format that data should be sent in, forms that "public face" of a web service. There will also be a backend script on the server side that can receive a request for one of the methods, and will return a response [3].

Once a programmer has read a service's WSDL file, they can use it to design a query to be sent to the service. This query is sent to the service in another XML format called SOAP (Simple Object Access Protocol) [2], a communication protocol designed to allow applications on the client side to access objects on the server side over common transfer protocols like HTTP, execute methods on those objects, and receive the results. Once a request has been expressed in SOAP syntax (after which it becomes known as a "SOAP envelope") it's sent to the URL specified in the WSDL document, where it is invoked on the backend script, and finally the response is sent back to the calling application.

The backend script is made up of three components: a listener to receive the message, a "proxy" to take that message and translate it into an action to be carried out, and program code to carry out that action. When the proxy component receives a request message from the listener, it must do three things:

1. "Deserialise" the XML-encoded message into a format usable by the native system.
2. Invoke the call.
3. Serialise the response back into a SOAP envelope so that it can be sent back to the caller.

[3] Xmethods is a directory of publicly available web services: http://www.xmethods.net/

Figure 1 below illustrates a high-level view of this process.

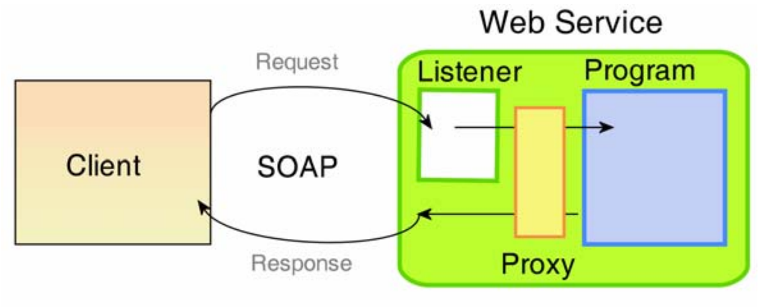


Figure 1. The elements of a web service.

The response to a call like this will generally be nothing more complicated than a single string value, wrapped in SOAP code for transport. A stock quote service for example, would take in a string for the quote requested, and simply return the resulting computation, "75.50" for example.

Therefore, this project must comprise two parts:

1. The first to take web forms and convert them into an equivalent WSDL description so that a programmer can see how to call various functions of our converted web service.

2. A second part is then needed to receive a SOAP Envelope, and convert this back into a normal GET or POST request to the original web form, so that the process works as seamlessly as any true web service. The output of the whole process will be the web page that the user would have received had they filled in the form, which should contain any and all relevant output from the "service".

2.3 WSDL

All WSDL files are structured into four sets of definitions [4]:

Element	Defines
<types>	The complex data types used by the web service, if any.
<message>	The messages used by the web service.
<portType>	The operations performed by the web service.
<binding>	The communication protocols used by the web service.

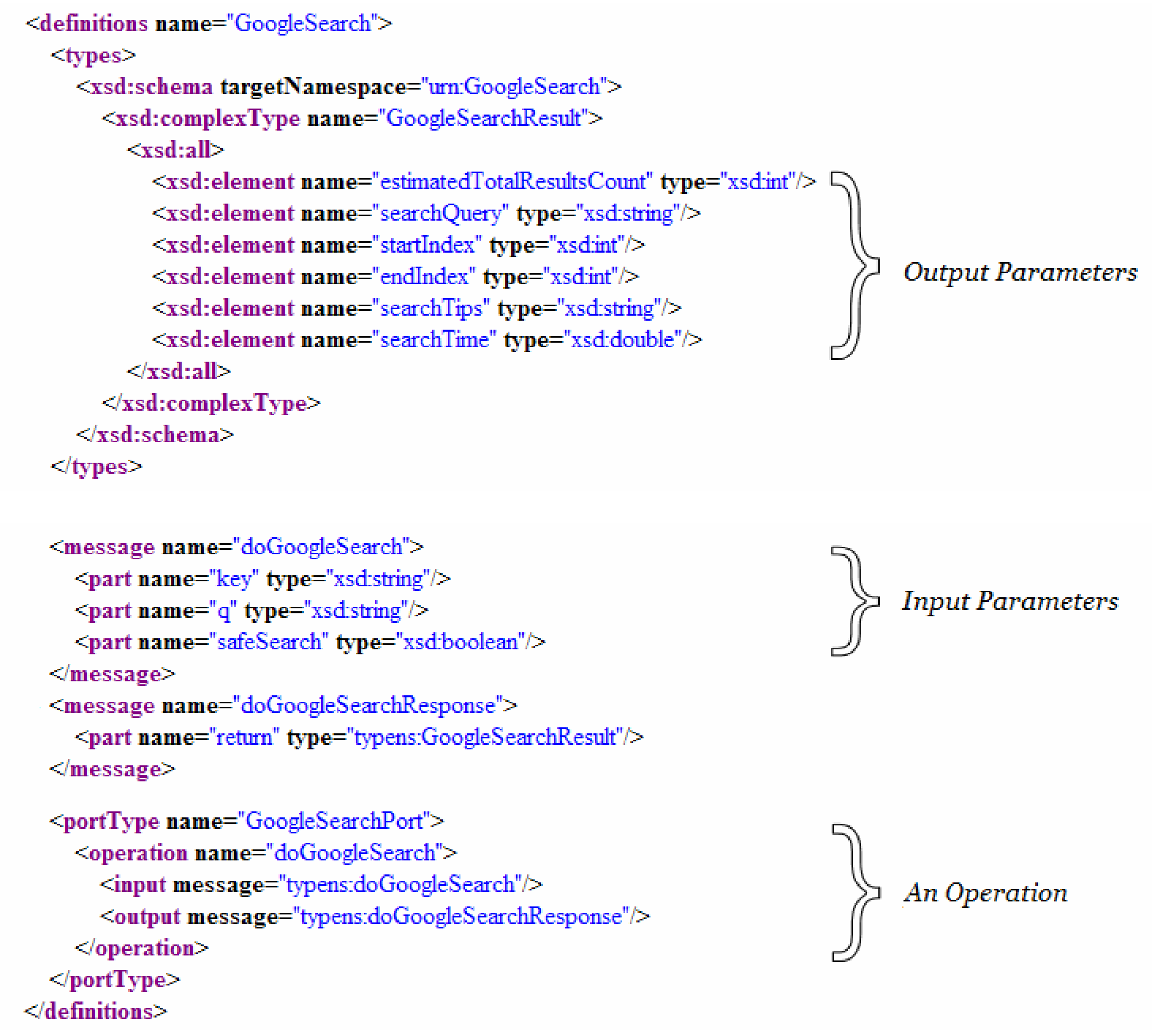
These elements correspond to aspects of a normal HTML form. The children elements of portType, `operation`, are mapped one-to-one to the forms on the page. Each WSDL file may contain multiple operation elements, so a single HTML file with multiple forms can be converted into a single WSDL file with multiple operations.

Nested inside each `operation` element are an `input` and an `output` element, describing the format of the data the service accepts and returns. Each of these is made up of many elements, called messages in WSDL. Each `message` describes an input element in the form. Composite objects can be constructed in the types section, but are not required for this project.

2.3.1 WSDL File Example

WSDL code can seem abstruse and over-complicated, a problem compounded by the introduction of namespaces so that multiple XML-based languages can be used together in the same document. It is often best to read WSDL from bottom to top to see the document through its multiple layers of abstraction.

As an example of a typical WSDL file's structure, Figure 2 shows an abridged version of Google's WSDL file[4]:



```
<definitions name="GoogleSearch">
  <types>
    <xsd:schema targetNamespace="urn:GoogleSearch">
      <xsd:complexType name="GoogleSearchResult">
        <xsd:all>
          <xsd:element name="estimatedTotalResultsCount" type="xsd:int"/>
          <xsd:element name="searchQuery" type="xsd:string"/>
          <xsd:element name="startIndex" type="xsd:int"/>
          <xsd:element name="endIndex" type="xsd:int"/>
          <xsd:element name="searchTips" type="xsd:string"/>
          <xsd:element name="searchTime" type="xsd:double"/>
        </xsd:all>
      </xsd:complexType>
    </xsd:schema>
  </types>

  <message name="doGoogleSearch">
    <part name="key" type="xsd:string"/>
    <part name="q" type="xsd:string"/>
    <part name="safeSearch" type="xsd:boolean"/>
  </message>
  <message name="doGoogleSearchResponse">
    <part name="return" type="typens:GoogleSearchResult"/>
  </message>

  <portType name="GoogleSearchPort">
    <operation name="doGoogleSearch">
      <input message="typens:doGoogleSearch"/>
      <output message="typens:doGoogleSearchResponse"/>
    </operation>
  </portType>
</definitions>
```

Figure 2. An extract from the WSDL for Google's Web Service

This segment of the file is concerned with an operation, "doGoogleSearch", which returns an output of type "doGoogleSearchResponse". The "doGoogleSearch" operation requires three input parameters: "key", "q", and "safeSearch". It will output a "GoogleSearchResult" object as its response. This object is a composite of a number of primitive elements, and has its attributes defined at the beginning of the file in the types section.

[4] The full version can be found at http://api.google.com/GoogleSearch.wsdl

2.3.2 WSDL Message/Element Correspondence

There are a dozen form input types defined in the HTML 4.01 specification [5], such as text inputs, radio buttons etc. The initial challenge faced was to design a representation of these inputs using the element types available for messages in WSDL. The WSDL specification uses a typing system called XML Schema Datatypes (XSD) [6], which is used in a number of other XML-derived languages to apply datatypes to document content.

XSD specifies many datatypes that will sound familiar to programmers, among them `string`, `int`, `boolean` and `float`. Each input in each form will need to be given one of these types.

A simplistic implementation would be to map all elements to the `string` datatype, since this is the most generic available and so any data can be passed as such. However, this affords little or no possibility of type-checking on the receiving end to ensure data interchange robustness. In the ideal case, the script receiving the XML data should be able to check it for conformance to the WSDL document and raise an error if data is sent in a differing format than was specified (for instance, if a date was required, but a floating point value was received).

The basic elements of a form – that is, single-line text inputs, hidden fields, password fields and multi-line text areas – can be mapped one-for-one as simple messages in WSDL, though further checking is required to choose a suitable datatype for each.

The complex elements – radio buttons, select boxes and submit buttons – necessitate the creation of new composite datatypes that represent a range of possible values. XSD includes the ability to create an enumerated set, of which one value can be chosen. This corresponds exactly to the behaviour of radio buttons and dropdown selection boxes.

2.4 Programming Language

It was decided early on that, since the project dealt with webpages, it made more sense to deploy the final program as a web application as opposed to a binary executable. This approach meant that a user could use the program from any machine, and didn't require any more complicated setup than an Internet connection and a web browser. The downside of using a script is that scripts running on a web server will be interpreted at runtime, which is generally slower than running a pre-compiled binary. However, since each execution of the script will at some point require connecting to a remote website and downloading a file – which due to the nature of the Internet will always take a variable amount of time – this is not considered a problem.

There are a range of languages that would have been suited for this program, such as PHP, ASP, Perl or Python. As there is a large amount of text processing involved, Perl was chosen due to its proficiency for parsing text with regular expressions. Perl is designed to be run remotely on a server as a CGI script, so it was easy to set up a web gateway to the project.

The Perl community maintain a repository of ready-made modules in their online archive CPAN[5] ("Comprehensive Perl Archive Network"), that anyone is free to use in their own projects. A number of modules were chosen to perform much of the heavy lifting, such as extracting the relevant HTML markup from a page. Therefore a lot of the critical operations of the program are performed by pre-existing code, much of which has been well-tested and is very stable. Modules could also be used to set up the requests and responses necessary when fetching webpages.

[5] CPAN can be searched at http://search.cpan.org/

2.5 Markup Quality

Before a webpage can be converted into its equivalent WSDL description, the HTML that makes it up must be analysed. Markup such as HTML and more specifically XML and XHTML has a set of guidelines that well-authored, “valid” code must adhere to [7]. These rules are generally simple constructs, such as closing tags being required for all elements, or attribute values needing to be encased in double quotes.

This affects this project because the Form2WSDL program will need to parse the page, primarily looking for form elements. Though the Perl module used to extract the form information is somewhat liberal in what quality of code it will accept, it is not hard to imagine a poorly-authored website that the module will not be able to work with.

For instance, as expressed in the HTML 4.01 specifications, the form element, which is used as a container for all of the input elements like checkboxes and radio buttons, is prohibited from having another form element nested inside it.

```
<form method="get" action="process.cgi">
  <input type="text" name="emailaddress" />
  <form method="get" action="resetdata.cgi">
    <input type="reset" />
  </form>
</form>
```

However, it must be taken into account that a page with this exact problem could be encountered ‘in the wild’. In this case there are a number of ways the code could be interpreted, from disregarding this second form, to inferring the closing tags of the previous form and treating the two forms as being separate. Ideally in this case, the latter option would be chosen.

To this end, raw HTML code should be cleaned up before it is passed to a form parser. This, to a large extent, removes the risk of abnormalities in how the pages fed to the parser are dealt with. “HTML Tidy” is a program that has been in use for years and is proven to convert invalid HTML or XHTML markup into cleanly authored, valid XML-based markup[6]. Not only does this safeguard make the form parsing more likely to succeed, but the parsing executes quicker on clean markup rather than having to normalise the HTML before trying any data extraction.

[6] HTML Tidy was originally written by Dave Raggett, one of the earliest specification writers at the W3C (World Wide Web Consortium). It has since been taken on as a project on SourceForge. http://www.w3.org/People/Raggett/tidy/

3 Form Parsing

3.1 Datatypes

Since the WSDL spec requires that all input and output messages are given a type, each corresponding HTML input needs to be analysed for clues as to what range of content they should accept. The parameter type `string` will be used as a baseline, and applied to all elements unless a better, more accurate alternative can be found.

Guessing at field types is known to be difficult [9, 10], and various methods have already been attempted, such as using Bayesian learning algorithms to create a program that makes smarter guesses as it encounters more and more forms. An approach like this was beyond the scope of this project, and so a simpler system based on strong evidence found in the HTML has been put into operation.

For instance, checkbox elements have a simple binary nature, possessing only two states – they are either checked or unchecked. These values correspond exactly to the `boolean` type, which can take the values true and false. The WSDL element outputted for a checkbox will look like this:

```
<part name="addtonewsletter" type="xsd:boolean"/>
```

The “`xsd:`” prefix is the namespace for XML Schema Datatypes. Each element from XSD must be prefixed like this to keep them separate from the native WSDL elements.

This is the simplest case; most of the other form element types pose their own problems. The accuracy of the program’s assertions on the datatype of text-based inputs like text areas is heavily dependant on extra information that may not be available in most forms, so they are often left with the default type. Text fields often come with very little description and so great care must be taken if a guess is to be made on a suitable parameter type, as choosing the wrong type could cause a well-meaning user to send XML data with the wrong type to the form, leading to skewed results.

All form inputs will have a `name` attribute, and in certain cases this will be enough to make an educated guess. For instance, the program might encounter this input:

```
<input type="text" name="quantity" />
```

It would not be unreasonable to assume a numeric value is what’s required in this case. However, semantics will come into play in many cases, and it will generally be better to play it safe and fall back on the `string` type rather than making an incorrect assumption.

Web designers will sometimes add a default value to a form input to imply the format that they want users to enter data in. Inputs possessing these pre-filled values imply salient details, and these prove the most useful indicator of an input’s suggested type. There are a number of built-in checks that will operate on the value of the input, if available.

The program could read the following input, and give it the type `date`.

```
<input type="text" name="start" value="13/07/1983" />
```

The program will recognise dates in the format DD/MM/YYYY, MM/DD/YYYY and YYYY/MM/DD, along with a range of different delimiters between the segments and give these messages the type `date`.

If an input’s value comes pre-filled as a number value like “1” or “2.5”, the script will give these messages the types `integer` and `float` respectively. It will also recognise commonly-used names like “amount” and “count”.

3.2 Complex Elements

The complex elements like selection drop-downs or radio buttons need to be dealt with separately to the ones already discussed. They can both be thought of as enumerated set of values, where only one value may be chosen. These choices are then given an overarching "base type" that they all adhere to. The program could encounter a select box in the HTML like so:

```
<select name="resultsperpage">
  <option value="10">Ten</option>
  <option value="20">Twenty</option>
  <option value="40">Forty</option>
</select>
```

This would be converted into an enumeration with three choices [8]. Some simple regular expressions are used to discover that all of the available options (the tokens set as each `option`'s value) are integers, and so the base type for this input is set to `positiveInteger` based on this data.

The resulting WSDL snippet for this select box looks like this:

```
<simpleType name="resultsperpage">
  <restriction base="xsd:positiveInteger">
    <enumeration value="10"></enumeration>
    <enumeration value="20"></enumeration>
    <enumeration value="40"></enumeration>
  </restriction>
</simpleType>
```

Another complex case is when a form has two or more submit buttons. In a web browser, the form will operate the same as a form with a single submit button, but the backend script will presumably behave differently depending on which is pressed. For instance, one could be labelled "continue" and another "back", and the script will branch; redirecting the user to the homepage if they select "back", and proceeding to the next stage of the checkout process if they select "continue".

If a form only has one submit button, it can safely be omitted from the WSDL conversion routine, since it does not represent a choice that a user has to make, and its inclusion is assumed. Otherwise, since the submit buttons are mutually-exclusive, they can also be modelled as an enumerated type, where only one of the set can be selected.

3.3 Further Element Parsing

Though most of the parameter-type guessing will be based on the element's type and pre-filled value, there is also scope for inferring information from nearby elements in the document's source code.

HTML 4.01 – and thus all varieties of XHTML 1.0 – includes an element called `<label>`, which is used to couple the textual description of an input with the input itself by means of the element's `id` value. This improves accessibility by making a strong connection between the input and its description, which are often decoupled by being placed in different table cells in order to create a particular visual layout.

An example of one use for `<label>`:

```
<tr>
  <td><label for="creditcard">Enter Credit Card number:</label></td>
  <td><input type="text" name="credit" id="creditcard" size="15" /></td>
</tr>
```

The connection between the `label` and `input` can be established programmatically, and the contents of the text node within the `label` element can be extracted. In this case the program would extract the string "`Enter Credit Card number:`" – which can easily be extrapolated to refer to an element that takes a string made exclusively of numbers (an `int`, in other words). Less obvious labels or forms that do not contain any label elements prove a much greater challenge to categorise correctly.

This approach, though the simplest to implement, is restricted somewhat by the need to manually enter each of the cases that the script can recognise, which encourages conservative choices. A better implementation would involve an aspect of machine learning to build up a knowledge base of which words can be used to help categorise an input.

3.4 Naming Elements

In WSDL each element must be given a name that is unique within the document. For the most part, these have had to be made up as no suitable name can be found in the HTML. The collection of `operation` elements, each of which correspond to a single form on the source page, are named FormNumberN, where N ranges upwards from 1. Forms can have a `name` attribute, though rarely used, which is given precedence if it can be found in the source.

A suitable name needed to be found for the WSDL file to uniquely identify a service. Existing web services use names like “GoogleSearchPort” and “AWSECommerceServicePortType”. Since a website’s URL is already a unique identifier for a site amongst others on the Internet, it makes sense to use that to uniquely identify its web service. Therefore, the domain name of a site is extracted and used to compose names like “ebayPort”.

These names combine to create WSDL in this form:

```
<portType name="ebayPort">
  <operation name="FormNumber1">
    <input message="Message1"/>
    <output message="MessageOut"/>
  </operation>
  <operation name="FormNumber2">
    <input message="Message2"/>
    <output message="MessageOut"/>
  </operation>
</portType>
```

Since all forms are thought of as outputting data in the same way, i.e. a webpage which for practical purposes we think of as a long string, every operation that gets defined returns this same `output` message “MessageOut”.

4 Implementation

The HTML to WSDL conversion script could be rapidly prototyped, another benefit of using Perl. Each aspect of the implementation is iterative, with the data being passed between methods and modified along the way. A simple web interface was built to allow a user to enter the URL of a page they want analysed for forms. This interface is available at

http://www.yourhtmlsource.com/projects/Form2WSDL/

4.1 Fetching Web Pages

Before the program can begin to process the HTML file, it must first fetch it from the web. Fetching the webpages is performed by a Perl module called LWP::UserAgent, which is the standard way to create either a robot or a "spider" (the distinction being that a spider will follow links to further pages).

When retrieving web pages from the server, a number of checks have to be performed on the file to make sure it is the file requested.

4.1.1 Following Redirections

Redirects occur all the time on the web, particularly on large corporate websites, where reorganisations and a change in the underlying scripting technology are commonplace. One may be required if a user types a commonly misspelled word as part of an URL, or entered the URL of a page that has been moved to another location on the server. Most redirects will occur transparently and will not be noticed by the end user [11, 12].

When presenting a report of the conversion it has performed, it is important that the program notifies the user that a redirection occurred along the way if this is the case, as the page the program ended up at may not be the page the user had meant to have converted. To this end, the program will follow all standard URL redirects and add a note to the report if the URL provided by the user and the URL returned to the program in the HTTP response differ in any way.

Many websites are not properly configured to perform redirects, and their web developers add a hack in the form of a meta element in the page's header section, like so:

```
<meta http-equiv="refresh" content="5; url=newurl.html">
```

This is a client-side redirect that operates in the browser on a timer (in the case above, the redirect will occur after 5 seconds of delay). However, when a robot visits the page, no redirect will occur. To get around this limitation, further parsing on the HTML page is performed in an attempt to find this meta element if it exists and extract the URL. If this URL is different to the originally requested URL, a note is added to the report suggesting that the user also performs a conversion on the referenced page.

4.2 HTML Parsing

Once the file has been fetched, it is saved to a temporary file locally, ready for processing. First it's tidied by HTML Tidy to clean up any obvious markup errors, so that the parsing modules have less work to do. In the early stages of this project, all of the HTML parsing was performed by a module called HTML::Form, which was found on CPAN. When run over a HTML file, it returns an array of the forms it finds, with each of these form objects containing an array of child inputs. This two-dimensional array is then iterated over to print out the simple form elements which require relatively little processing.

When it came time to convert the complex elements, the data returned by HTML::Form was found to be insufficient to create a correct representation of these elements. For an input with multiple choices, like a set of radio buttons, only the first input was returned, meaning that an enumerated set could not be built, as this requires data about each of the inputs in the group.

Further data extraction was necessary, so another module was employed. HTML::TreeBuilder, again found on CPAN, takes the HTML page as input and builds a tree structure from it, which can then be traversed in pre-order looking for elements that meet a certain set of criteria [13]. This makes it easy to find a certain element, and then extract its siblings from the tree to build a composite element.

Since an array of elements (with their input types) has already been created through HTML::Form, this is expanded by iterating through it, finding the names of the elements that require further extraction, and searching the tree for these elements, which are then placed into an array, ready to be printed.

This was done so that the original work did not have to be completely scrapped, though it's not the most efficient manner of parsing the page, since it uses two separate data structures to store data about some of the same elements. In this case code simplicity won out over efficiency. Had HTML::TreeBuilder been used for all form extraction, the program would have consumed fewer resources. Again, as the script is generally only run once and will involve fetching a webpage, execution speed was not made a high priority.

Once all of the forms had been parsed, the complete WSDL for all of the forms on the page is written as output.

4.3 Invoking the Web Service

Now that suitable WSDL code is viewable by a programmer, all the information required to begin creating SOAP requests to this service is available. The final step in this project was building an invoking script that could respond to these requests, convert them into calls to the original form, and then pass back the results of the form as the response to the SOAP request.

Much of the code from the original conversion component of the project could be re-used for this part. A call to this new script would contain the URL of a page, and the unique identifier of a form on that page (something like "FormNumber2"). Once the script has these two things, it fetches the requested page, and again parses this page for forms (this time using the simpler HTML::Form module exclusively, as there is no need to know everything about each input; we only need to know each input's value).

The SOAP envelope sent to the invoking script is deserialised by another commonly-used Perl module, SOAP::Lite. The SOAP request sent to the script will contain a value for each of the inputs in the form. Parsing the page for forms means we have an internal representation of the form in memory. The next step is to change the values of each of the inputs in this form to correspond to the values that were sent in the SOAP request, using methods contained in HTML::Form. Once the form has been modified in this way, it is submitted with these values, just as if a user had filled in these values and clicked the submit button in their browser.

This results in a request object that can be passed to an LWP::UserAgent so that the resulting page can be fetched. This page will be the results of the form, and so will contain the theoretical "output" of the web service. This is then serialised and sent back to the caller using SOAP::Lite, so that they may use this data however they wish.

5 Testing

To test the script's effectiveness in parsing and converting real-life forms into their WSDL equivalents, it has been run on a number of popular websites.

Complex online processes, such as buying airline tickets are, in general, not possible to test like this, as the forms a user must use to complete the transaction are split over a series of pages. Each form is then just a single step in a larger process, with the resultant page containing another form.

5.1 A Search Engine

Example site: All the Web, http://www.alltheweb.com/

A search engine's query form will generally consist of nothing more than a basic interface that will return a page of webpage results. This is a popular implementation of web services, so much so that Google restricts the number of accesses to its service to one thousand a day. The basic elements of a search interface do not change much from site to site: there is a text input box for search keywords, possibly a selection box to narrow the scope of the search to a particular subset of topics, like newsgroup posts for example, and a single submit button.

The All the Web search engine's interface is spare, consisting of a text input, a number of hidden fields, a submit button and two radio buttons to choose your language. The script converts all text inputs and hidden fields to messages with type `string`, which is the most appropriate. The radio buttons are only applicable to visitors using web browsers, as they use HTTP cookies to store a user's preference. This doesn't affect the Form2WSDL converter.

5.2 A Query-based Web Application

Example site: Universal Currency Converter, http://xe.com/ucc/

This site sports a very simple interface for performing a currency conversion between two of a large range of world currencies, a perfect application for a web service. There is a text input (initially set to "1"), two select boxes with an identical range of options for the currencies, and a single submit button. There is limited guide text which mentions the words "amount" and "currency", but these are in separate table cells and aren't linked in any way to the inputs they correspond to, so the program is blind to them.

The program guesses that the first input is of type `int`. A better guess would probably have been `double`, since a user may want to convert a value like "2.37". This would've been improved had the nearby text been marked up in some way that it could be inferred that it described this input.

The two selection boxes are converted correctly into two enumerated sets, with the base type `string`, since the values are of the form "EUR", "USD" etc. Since there is only one submit button, this doesn't turn up in the WSDL output, as it doesn't represent a choice the user has to make.

5.3 The Ideal Case

Site: Form2WSDL Test Page, http://www.yourhtmlsource.com/projects/Form2WSDL/testing/

Finally, the script was run over this page that was made to test the program during development. It represents an ideal form: one in which every input has a logical name value, and has a pre-filled value for each, all with properly marked-up labels. On this form the program's success rate is almost 100%.

6 Conclusions

Overall, the conversion from HTML forms to WSDL descriptions is considered a success.

Like many XML-derived languages, WSDL can feel verbose, over-complicated and obfuscated. Many companies that would benefit from having a usable web service to allow their customers to access details about their products have not yet made a service available, whether that is because they are unaware of the opportunity or have found the technical challenge too great.

It is the author's view that this program would serve as an excellent training tool for those learning the syntax of WSDL. Prototypical WSDL files can be generated from pre-existing web forms and then manually tailored to the system by hand. Though an automated tool like this can never fully replace a properly-authored WSDL file by an engineer of the system, the WSDL documentation that is output serves as an excellent starting point for further refinement.

In effect, the Form2WSDL program could perform the same function as WYSIWYG (What You See is What You Get) HTML editors among web designers – they make it easier for web application programmers to start creating their own web services, without having to know the language inside-out.

Further work in this area could take the form of refining the outputted WSDL code. Various machine learning algorithms would help the program make better guesses on suitable data types for inputs. There is much scope for using word proximity to infer classifications for inputs that are otherwise going to receive the type `string`. The SOAP data that is sent to the service could also be cross-validated against the WSDL file to make sure that all data has been sent in the right data type.

7 References

[1] E. Christensen, et al. *Web Services Description Language (WSDL) 1.1,* 15 March 2001.
URL: http://www.w3.org/TR/wsdl

[2] M. Gudgin, et al. *Simple Object Access Protocol,* 24 June 2003.
URL: http://www.w3.org/TR/soap12-part1/

[3] E. Cerami. *Web Services Essentials: Distributed Applications with XML-RPC, SOAP, UDDI & WSDL.* O'Reilly, 2002.

[4] W3Schools. *The WSDL Document Structure.*
URL: http://www.w3schools.com/wsdl/wsdl_documents.asp

[5] D. Raggett, A. Le Hors, I. Jacobs. *Form elements in the HTML 4.01 specification*, 24 December 1999.
URL: http://www.w3.org/TR/REC-html40/interact/forms.html#h-17.3

[6] P. V. Biron, A. Malhotra. *XML Schema Part 2: Datatypes Second Edition,* 28 October 2004.
URL: http://www.w3.org/TR/xmlschema-2/

[7] W3C. *XHTML 1.0 The Extensible HyperText Markup Language*, 1 August 2002.
URL: http://www.w3.org/TR/xhtml1/#diffs

[8] P. V. Biron, A. Malhotra. *XSD Enumeration type,* 28 October 2004
URL: http://www.w3.org/TR/xmlschema-2/#dt-enumeration

[9] N. Kushmerick. *Learning to invoke Web Forms*, 2003.
URL: http://www.cs.ucd.ie/staff/nick/home/research/download/kushmerick-odbase2003.pdf

[10] A. Heβ, N. Kushmerick. *Learning to Attach Semantic Metadata to Web Services,* 2003. In *Proc. Int. Semantic Web Conf..*

[11] R. Fielding, et al. *Header Field Definitions in the Hypertext Transfer Protocol (HTTP 1.1) specification*, February 1999.
URL: http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html

[12] R. S. Engelschall. Apache *URL Rewriting Guide*, December 1997.
URL: http://httpd.apache.org/docs-2.0/misc/rewriteguide.html

[13] Sean M. Burke. *HTML::Element*, September 15, 2003.
URL: http://search.cpan.org/~sburke/HTML-Tree-3.18/lib/HTML/Element.pm