Service Class Driven Dynamic Data Source Discovery with DynaBot[1]

Daniel Rocco
Department of Computer Science
University of West Georgia
Carrollton, GA 30118 USA
drocco@westga.edu

James Caverlee
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332 USA
caverlee@cc.gatech.edu

Ling Liu
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332 USA
lingliu@cc.gatech.edu

Terence Critchlow
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
Livermore, CA 94551 USA
critchlow1@llnl.gov

ABSTRACT:

Dynamic Web data sources – sometimes known collectively as the Deep Web – increase the utility of the Web by providing intuitive access to data repositories anywhere that Web access is available. Deep Web services provide access to real-time information, like entertainment event listings, or present a Web interface to large databases or other data repositories. Recent studies suggest that the size and growth rate of the dynamic Web greatly exceed that of the static Web, yet dynamic content is often ignored by existing search engine indexers owing to the technical challenges that arise when attempting to search the Deep Web. To address these challenges, we present DYNABOT, a service-centric crawler for discovering and clustering Deep Web sources offering dynamic content. DYNABOT has three unique characteristics. First, DYNABOT utilizes a service class model of the Web implemented through the construction of service class descriptions (SCDs). Second, DYNABOT employs a modular, self-tuning system architecture for focused crawling of the Deep Web using service class descriptions. Third, DYNABOT incorporates methods and algorithms for efficient probing of the Deep Web and for discovering and clustering Deep Web sources and services through SCD-based service matching analysis. Our experimental results demonstrate the effectiveness of the service class discovery, probing, and matching algorithms and suggest techniques for efficiently managing service discovery in the face of the immense scale of the Deep Web.

KEY WORDS:
Service discovery, Web crawling, Dynamic Web data, Deep Web

INTRODUCTION

The World Wide Web is the product of two unique approaches to document publication. The traditional or "static" Web consists of documents materialized in the secondary storage of server systems that are hyperlinked to other Web documents. These documents are generally accessible to unauthenticated users and automated agents like search engine crawlers. The dynamic or "Deep Web," in contrast, refers to the dynamic collection of Web documents that are created as a

[1] The DynaBot project homepage is http://www.cc.gatech.edu/projects/disl/specialProjects/Dynabot.html.

direct response to some user query. Deep Web services provide access to real-time information, like entertainment event listings, or present a Web interface to large databases or other data repositories. Recent studies suggest that the size and growth rate of the dynamic Web greatly exceed that of the static Web (Lawrence & Giles, 1998; Lawrence & Giles, 1999). Estimates suggest that the practical size of the Deep Web may be greater than 550 billion individual documents (Bergman, 2003). More than half of the content of the Deep Web resides in topic-specific databases, many of which are made available through Web services. A full ninety-five percent of the Deep Web is publicly accessible information that is not subject to fees or subscriptions.

Dynamic content is often ignored by existing search engine indexers owing to the technical challenges that arise when attempting to search the Deep Web. The most significant challenge is the philosophical difference between the static and Deep Web with respect to how data is stored: in the static Web, data is stored in documents while in the dynamic Web, data is stored in databases or produced as the result of a computation. This difference is fundamental and implies that traditional document indexing techniques, which have been applied with extraordinary success on the static Web, are inappropriate for the Deep Web. Related to the data storage issue is the problem of data retrieval, since static Web documents are retrieved via simple HTTP calls while dynamic Web documents often reside behind form interfaces that are impenetrable to traditional crawlers. Finally, Deep Web sources tend to be more domain-focused than their static Web counterparts. While there is much to be gained from discovering and clustering Deep Web sources, any significant exploration of the Deep Web will require techniques that exploit service-oriented functionality through intelligent analysis of search forms and result samples.

With these challenges in mind, we present DYNABOT, a service-centric crawler for discovering and clustering Deep Web sources. DYNABOT has three unique characteristics. First, DYNABOT utilizes a service class model of the Web implemented through the construction of service class descriptions (SCDs). Second, DYNABOT employs a modular, self-tuning system architecture for focused crawling of the Deep Web. Third, DYNABOT incorporates methods and algorithms for efficient probing of the Deep Web and for discovering and clustering Deep Web sources and services through SCD-based service matching analysis.

We demonstrate the capability of DYNABOT through the BLAST [**B**asic **L**ocal **Al**ignment **S**earch **T**ool] service discovery scenario. Our initial experimental results are very encouraging – demonstrating up to 73% success rates of service discovery and showing how the incorporation of service clues into the search process may improve service matching throughput. These results suggest an opportunity for efficient service discovery in the face of the large and growing number of web services. The DYNABOT prototype has been successfully deployed by Lawrence Livermore National Lab for use in aiding bioinformatic service discovery and integration, and its further development and testing is continuing as part of the LDRD project; the goal of this project is to provide scientists with access to hundreds of data sources through a single, intuitive interface, thereby simplifying their interaction with data and enabling them to answer more complex questions than currently possible.[2]

The remainder of the paper is organized as follows. We first describe our service class model and the construction of service class descriptions. We then outline the architectural design of DYNABOT with a focus on system-level design and development issues, including our Deep Web probing methodology and the SCD-based service matching algorithms. We then present our initial experimental results that demonstrate the effectiveness and scalability of DYNABOT for

[2] http://disl.cc.gatech.edu/LDRD/

discovering domain specific Deep Web sources and services. We then conclude the paper with a summary of related work and a discussion of open research issues.

THE SERVICE CLASS MODEL

Research on DYNABOT for automatically discovering and classifying web services is motivated by the need to fill the gap between the growth rate of web services and the rate at which current tools can interact with these services. Given a domain of interest with defined operational interface semantics, can we provide superior service identification, classification, and integration services than the current state-of-the-art?

To facilitate the domain-specific discovery of web services, we introduce the concept of service classes. We model a service provider *S* as a provider of *k* services $s_1, \ldots, s_k$ ($k \geq 1$). The service class model views the spectrum of web services as a collection of *service classes*, which are services with related functions.

Definition 1: A *service class* is a set of web services that provide similar functionality or data access.

The definition of the desired functionality for a service class is specified in a *service class description*, which defines the relevant elements of the service class without specifying instance-specific details. The service class description articulates an abstract interface and provides a reference for determining the relevance of a particular service to a given service class. The service class description is initially composed by a user or service developer and can be further revised via automated learning algorithms embedded in the DYNABOT service probing and matching process.

Definition 2: A *service class description* (SCD) is an abstract description of a service class that specifies the minimum functionality that a service *s* must export in order to be classified as a member of the service class. An SCD is modeled as a triple: $SCD = \langle T, G, P \rangle$, where *T* denotes a set of *type definitions*, *G* denotes a *control flow graph*, and *P* denotes a set of *probing templates*.

The service class model supports the web service discovery problem by providing a general description of the data or functionality provided. A service class description encapsulates the defining components that are common to all members of the class and provides a mechanism for hiding insignificant differences between individual services, including interface discrepancies that have little impact on the functionality of the service. In addition, the service class description provides enough information to differentiate between a set of arbitrary web services.

As a continuing example, consider the problem of locating members of the service class `Nucleotide BLAST`. Nucleotide BLAST services provide complex similarity search operators over massive genetic sequence databases. BLAST services are especially important to bioinformatics researchers. The relevant input features in this service class are a string input for specifying genetic sequences, a choice of nucleotide databases to search, and a mechanism for submitting the genetic sequence query to the appropriate server. The relevant output is a set of sequence matches. Note that this description says nothing about the implementation details of any particular instance of the service class; rather, it defines a minimum functionality set needed to classify a service as a member of the `Nucleotide BLAST` service class. SCDs may also be defined for user-specified classes like `Keyword-Based Search Engines`, `Stock`

`Tickers`, or `Hotel Reservation Services`. The granularity of each SCD is subject to the user needs.

Our initial prototype of the DYNABOT service discovery system utilizes a service class description composed of three building blocks: type definitions, a control pattern, and a set of probing templates. The remainder of this section describes each of these components with illustrative examples.

Type Definitions

The first component of a service class description specifies the data types $t \in T$ that are used by members of the service class. Types are used to describe the input and output parameters of a service class and any data elements that may be required during the course of interacting with a service. The DYNABOT service discovery system includes a type system that is modeled after the XML Schema (Fallside, 2001) type system with constructs for building atomic and complex types. This regular expression-based type system is useful for recognizing and extracting data elements that have a specific format with recognizable characteristics. Since DYNABOT is designed with a modular, flexible architecture, the type system is a pluggable component that can be replaced with an alternate implementation if such an implementation is more suitable to a specific service class.

```
<type name="DNASequence"
      type="string"
      pattern="[GCATgcat-]+" />

<type name="AlignmentSequenceFragment" >
    <element name="AlignmentName"
             type="string"
             pattern="[:alpha:]+:" />
    <element type="whitespace" />
    <element name="start-align-pos"
             type="integer" />
    <element type="whitespace" />
    <element name="Sequence"
             type="DNASequence" />
    <element type="whitespace" />
    <element name="end-align-pos"
             type="integer" />
</type>
```

Figure 1: Nucleotide BLAST: type definitions

The regular expression type system provides two basic types, atomic and complex. *Atomic types* are simple valued data elements such as strings and integers. The type system provides several built in atomic types that can be used to create user-defined types by restriction. Atomic types can be composed into *complex types*, which are formed by composition of basic types into larger units.

The `DNASequence` type in Figure 1 is an example of an atomic type defined by restriction in the nucleotide BLAST service class description. Each type has a type name that must be unique within the service class description. Atomic types include a base type specification (e.g. `type="string"`) which can reference a system-defined type or an atomic type defined elsewhere in the service class description. The base type determines the characteristics of the type that can be further refined with a regular expression pattern that restricts the range of values acceptable for the new type. More intricate types can be defined using the complex type definition, which is composed of a series of elements. Each element in a complex type can be a reference to another atomic or complex type or the definition of an atomic type. List definitions are also allowed using the constraints `minOccurs` and `maxOccurs`, which define the expected cardinality of a particular sub-element within a type. The `choice` operator allows types to contain a set of possible sub-elements from which one will match. Figure 1 shows the declaration for a complex type that recognizes a nucleotide BLAST result alignment sequence fragment, which is a string similar to:

```
Query:   280 TGGCAGGCGTCCT 292
```

The above string in a BLAST result would be recognized as an `AlignmentSequenceFragment` by the type recognition system during service analysis.

Control Flow Graph

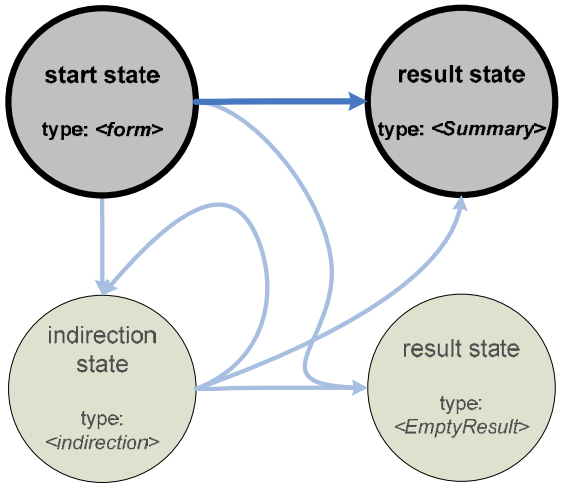
Due to the complexity of current services, we model the underlying control flow of the service with a control flow graph. In the BLAST scenario, each request to the server may have multiple possible response types depending on the current server and data availability, as well as the user permissions. For example, a query that results in a list of genetic sequences under normal load conditions may result in a completely different `Unavailable` response, or, perhaps, an intermediate `Wait 30 Seconds` response until the list of resulting genetic sequences is returned. By defining a control flow graph to capture these different scenarios, we can help guide the choice of the appropriate semantic analyzer for use on each response.

A service class description's *control flow graph* is a directed graph $G = (E,V)$, consisting of a set of state nodes V connected by directed edges $e \in E$. The state nodes in the graph represent control points that correspond to pages expected to be encountered while interacting with the service. Each state $s \in V$ has an associated type $t \in T$. The directed edges depict the possible transition paths between the control states that reflect the expected navigational paths used by members of the service class.

Data from a web service is compared against the type associated with the control flow states to determine the flow of execution of a service from one state to another. Control proceeds from a start state through any intermediate states until a terminal (result) state is reached. The control flow graph defines the expected information flow for a service and gives the automated service analyzer, described in the Service Analyzer section, a frame of reference for comparing the responses of the candidate service with the expected results for a member of the service class. In order to declare a candidate service a match for the service class description, the service analyzer must be able to produce a set of valid state transitions in the candidate service that correspond to a path to a terminal state in the control flow graph.

Returning to our continuing example, Figure 2 provides an illustration of a service class control flow graph for a `Nucleotide BLAST` web service. The control flow graph has four state nodes

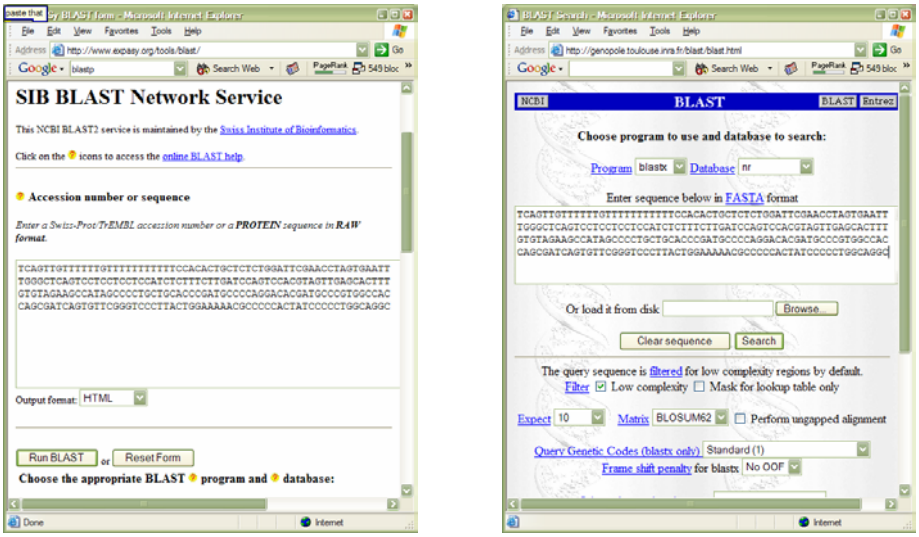
that consist of a state label and a data type. The control flow graph has a single start state that defines the input type a class member must contain. To be considered a candidate `Nucleotide BLAST` service, the service must produce either a single transition to a results summary state (as is highlighted in Figure 2) or a series of transitions through indirection states before reaching the summary state. This last point is critical – many web services go beyond simple query-response control flow to include complex control flow. In the case of `Nucleotide BLAST`, many services produce a series of intermediate results as the lengthy search is performed.



State	Transition States
start	**summary result** empty result indirection
indirection	indirection summary results empty result

Figure 2: Nucleotide BLAST: control flow graph

DYNABOT uses the service class description control flow graph to determine that a candidate is a member of a particular service class and to guide the choice of semantic analyzer for finer-grained analysis. So, when DYNABOT encounters a `Protein BLAST` service that resembles a `Nucleotide BLAST` service in both interface and the form of the results but differs in control flow, it will use the control flow analysis to appropriately catalog the service as a `Protein BLAST` service and then invoke domain-specific semantic analyzers for further analysis.



(a) (b)

Figure 3: Example simple (a) and complex (b) search forms

Probing Templates

The third component of the service class description is the set of probing templates *P*, each of which contains a set of input arguments that can be used to match a candidate service against the service class description and determine if it is an instance of the service class. For example, Figure 3 shows two example search form interfaces for BLAST search. The example on the left has a relatively simple interface, including a text box for entering a DNA sequence and a single submit button. The example on the right is more complex, with additional search arguments for the specific BLAST program, the BLAST database, options for filtering query results, and so on.

Our goal is to define a probing template for this search interface that is generic enough to match both example interfaces, but not so broad as to match non-BLAST sites. Probing templates are composed of a series of arguments and a single result type. The arguments are used as input to a candidate service's forms while the result type specifies the data type of the expected result. Figure 4 shows an example probing template used in a service class description. The probing template example shows an input argument and a result type specification; multiple input arguments are also allowed. The attribute `required` states whether an argument is a required input for all members of the service class. In our running example, all members of the `Nucleotide BLAST` service class are required to accept a DNA sequence as input. The argument lists the type of the input as well as a value that is used during classification. The optional `hints` section of the argument supplies clues to the service classifier that help select the most appropriate input parameters on a web service to match an argument. Finally, the output result specifies the response type expected from the service. All the types referenced by a probing template must have type definitions defined in the type section of the SCD.

```
<example>
  <arguments>
    <argument required="true">
      <name>sequence</name>
      <type>DNASequence</type>
      <hints>
        <hint>sequence</hint>
        <inputType>text</inputType>
      </hints>
      <value>TTGCCTCACATTGTCACTGCAAAT
             CGACACCTATTAATGGGTCTCACC
      </value>
    </argument>
  </arguments>

  <result type="SummaryPage" />
</example>
```

Figure 4: Nucleotide BLAST: probing template

The argument hints specify the expected input parameter type for the argument and a list of likely form parameter names the argument might match. Multiple name hints are allowed, and each hint is treated as a regular expression to be matched against the form parameters. These hints are written by domain experts using their observations of typical members of the service class. For example, a DNA sequence is almost always entered into a text input parameter, usually with "sequence" in its name. The DNA Sequence argument in a `Nucleotide BLAST` service class therefore includes a name hint of "sequence" and an input hint of "text."

DYNABOT SOFTWARE DESIGN

The problem of discovering and analyzing web services consists of locating potential services and determining their service interface and capabilities. The current approach to service discovery is to query or browse a known service registry, such as the emerging UDDI directory standard [http://www.uddi.org/]. However, discovery systems that rely solely on service registries have several drawbacks as discussed in the Introduction. In contrast, DYNABOT relies on a complementary approach that relies on domain-specific service class descriptions powered by an intelligent Deep Web crawler. This approach is widely applicable to the existing Web, removes the burden of registration from service providers, and can be extended to exploit service registries to aid service discovery.

Architecture

The first component of DYNABOT is its service crawler, a modular web crawling platform designed to discover those web services relevant to a service class of interest. The discovery is performed through a service class description-based service location and service analysis process. The DYNABOT service crawler starts its discovery process through a combination of visiting a set of given UDDI registries and a robot-based crawling of the Deep Service Web. By seeding a crawl with several existing UDDI registries, DYNABOT may identify candidate services that

match a particular user-specified service class description. These matching services need not be pre-labeled by the registry; the DYNABOT semantic analyzers will determine the appropriate classification based on the provided service class descriptions.

By expanding the discovery space through focused crawling of the Deep Web of services, DYNABOT may discover valuable services that are either not represented in current registries or are overlooked by current registry discovery tools. Recent estimates place the practical size of the Deep Web of web-enabled services at over 300,000 sites offering 1.2 million unique services, with the number of sites more than quadrupling between 2000 and 2004 (Chang et al., 2004). Deep Web services provide query capability that ranges from simple keyword search to complex forms with multiple options.

DYNABOT utilizes an advanced crawler architecture that includes standard crawler components like a URL frontier manager (Heydon & Najork, 1999), network interaction modules, global storage and associated data managers, and document processors, as well as a DYNABOT-specific service analyzer, which analyzes the candidate services discovered through focused crawling and determines if a service is related to a particular domain of interest by matching it with the given SCD, such as the `Nucleotide BLAST` service class description.

Crawling the Web for dynamic data sources shares many features with standard Web crawling. Table 1 compares the features of three classes of Web crawlers: simple, basic crawlers, advanced crawlers, and our own DYNABOT crawler. The components that make up a crawler are divided among three major component groups: network interaction modules, global storage and associated data managers, and document processing modules. The simplest crawlers require mechanisms for retrieving documents and determining if a particular URL has been seen. More advanced crawlers will include features like mirror site detection and trap avoidance algorithms. DYNABOT utilizes an advanced crawler architecture for source discovery and adds a document processor that can determine if a dynamic Web source is related to a particular domain of interest. Figure 5 shows the architecture of DYNABOT.

	Network Interaction	**Data Management**	**Processing Modules**
Basic Crawler	name resolver, document retrieval	URL frontier, visited list	link extractor, keyword index builder
Advanced Crawler	caching name resolver, multithreaded document retrieval	URL frontier, visited list, document cache	link extractor, keyword index builder, mirror detector, trap detector
DynaBot Crawler	caching name resolver, multithreaded document retrieval	URL frontier, visited list, document cache	link extractor, service class analyzer

Table 1: Web crawler feature comparison

Network Interaction. The network interaction modules handle the process of retrieving documents from the Internet, including the resolution of domain names. The significant costs associated with accessing data over the Internet can be amortized using multithreading to handle multiple requests simultaneously. This technique minimizes the penalty incurred when attempting to access a document from a server that is down or extremely slow. A second optimization technique is to cache DNS requests to reduce the number of network interactions needed and thereby improve document throughput. The effectiveness of DNS caching is due to

the high degree of domain locality in Web hyperlinks, which often reference different documents on the same server. In such cases, DNS name resolution is done once for all documents in the domain.

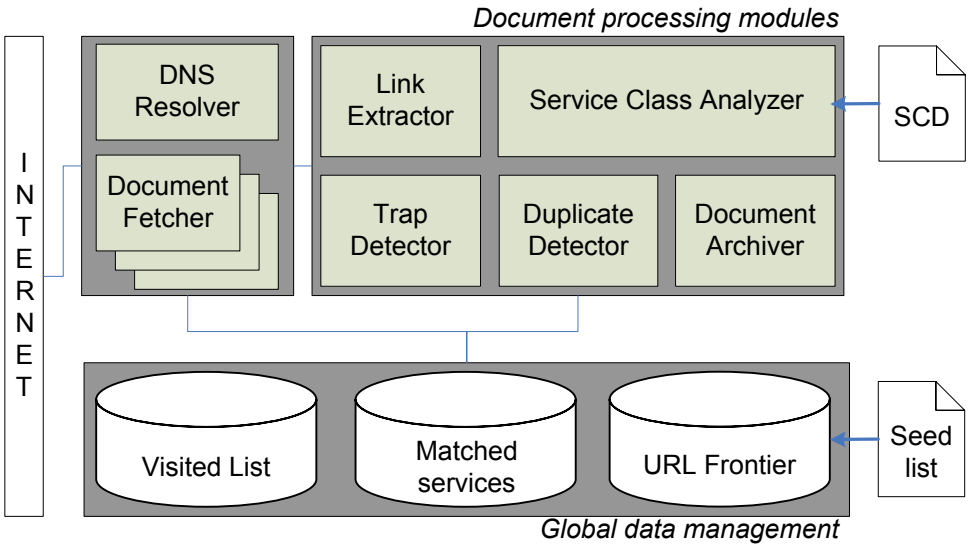


Figure 5: DynaBot System Architecture

Global Data Management. The crawler's global data management and storage components include the URL frontier and the visited list, which are responsible for tracking URLs the crawler has yet to visit as well as those that have already been processed. Managing the immense amount of data that a Web crawler will encounter is a technically interesting problem due to the sheer size of the Web. Global data is typically stored on disk, with caches used to reduce the disk storage penalty. Some crawlers will also store archives of crawled pages.

Processing Modules. The network interaction and global storage components are united by the processing modules, which initiate document retrieval, update the global storage with visited and new links, and perform any document processing required by the crawler's designated task. Processing modules are pluggable components that allow the crawler to be reconfigured for new tasks easily. A typical Web crawler includes a link extraction module, which extracts hyperlinks from the document, converts any relative links to their absolute form, and inserts them into the URL frontier. More sophisticated crawlers will include modules such as a duplicate content or mirror detector and trapped detection facilities to prevent the crawler from becoming ensnared in crawler traps.

Service Analyzer

The task of determining the capabilities and interface of dynamic Web sources is assigned to the Service Analyzer, a processing module of the DYNABOT crawler. The process of source discovery begins with the construction of the service class description, which directs the probing operations used by the service analyzer to determine the relevance of a Web source. The *service analyzer* consists of form filter and analyzer, an extension mechanism, a query generator, a query prober, and a response matcher.

Overview. When the processor encounters a new site to test, its first task is to invoke the *form filter*, which ensures that the candidate source has a form interface (Figure 6(1)). The second step (2) is to extract the set of forms from the page, load the service class description, and load any auxiliary modules specified by the service class description (3). The query generator (4) produces a set of query probes which are fed to the query probing module (5). Responses to the query probes are analyzed by the response matcher (6). If the query response matches the expected result from the service class description, the Web source has matched the service class description and a source capability profile (7) is produced as the output of the analysis process. The capability profile contains the specific steps needed to successfully query the Web source. If the probe was unsuccessful, additional probing queries can be attempted.

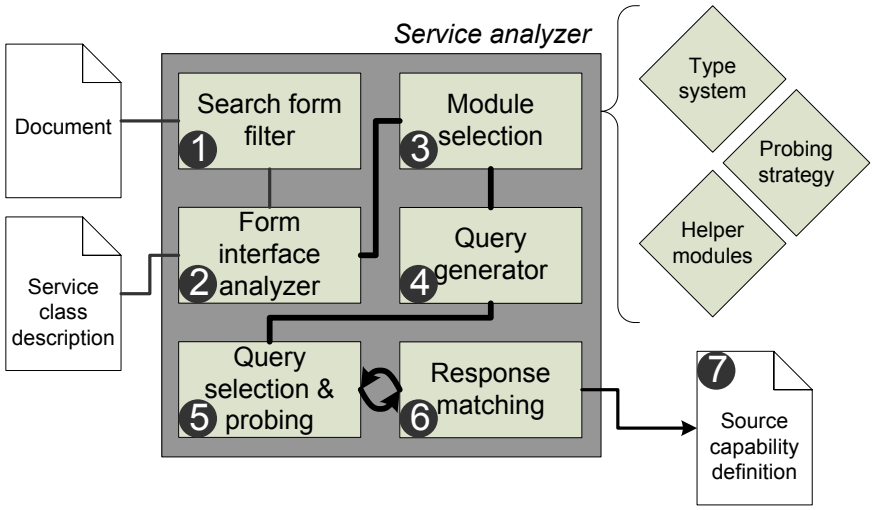


Figure 6: DynaBot Service Analyzer

Definitions. The process of analyzing the Web source begins when the crawler passes a potential URL for evaluation to the source analysis processing module. A source S for our purposes consists of an initial set of forms F. Each form $f \in F$, $f = (P,B)$ is composed of a set of parameters $p \in P$, $p = (t,i,v)$ where t is the *type* of the parameter, such as checkbox or list, i is the parameter's *identifier*, and v is the *value* of the parameter. The form also contains a set of buttons $b \in B$ which trigger form actions such as sending information to the server or clearing the form. The source S may specify a default for each parameter value v.

The process of *query probing* involves manipulating a source's forms to ascertain their purpose with the ultimate goal of determining the function of the source itself. Although the expected inputs and purpose of each of the various parameters and forms on a source is usually intuitive to a human operator, an automated computer system cannot rely on human intuition and must determine the identity and function of the source's forms algorithmically. The query probing component of the DYNABOT service analyzer performs this function. Our query prober uses induction-based reasoning with examples: the set of examples $e \in E$ is defined as part of the service class description. Each example e includes a set of arguments $a \in A$, $a = (r,t,v)$, where r indicates if the example parameter is required or optional, t is the type of the parameter, and v is the parameter's value.

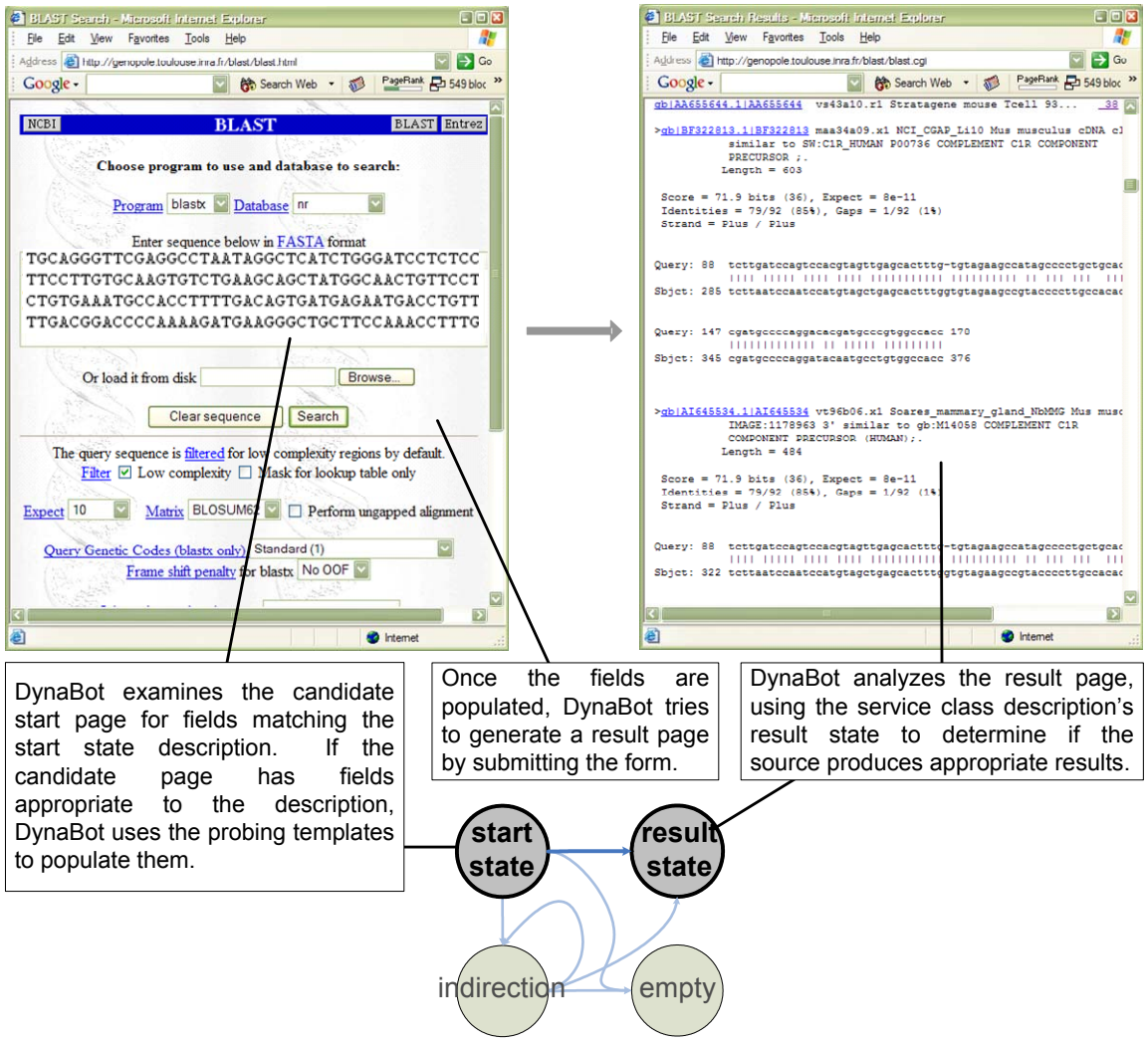


Figure 7: DynaBot Analysis of a Matching Web Source

Figure 7 illustrates the query probing process on a typical Web-based nucleotide BLAST service. Using the probing template as a guide, DYNABOT populates the site's forms as it attempts to determine the classification of the site. Once the fields have been populated, DYNABOT submits the form and checks to see if the result page matches the service class description's result state; on success, the service is classified as a match. If the service does not match, the probing process is repeated until a successful form configuration is found or the set of query probes is exhausted.

Form Filter and Analyzer. The *form filter* processing step helps to reduce the service search space by eliminating any source *S* that cannot possibly match the current service class description. In the filtration step, shown in step 1 of Figure 6, form filter eliminates any source *S* from consideration if the source's form set is empty, that is $F = \emptyset$. In form analysis, shown in step 2, the service class description will be compared with the source, allowing the service analyzer to eliminate any forms that are incompatible with the service class description. Algorithm 1 sketches the steps involved in the form filter process.

Algorithm 1: Form Filter

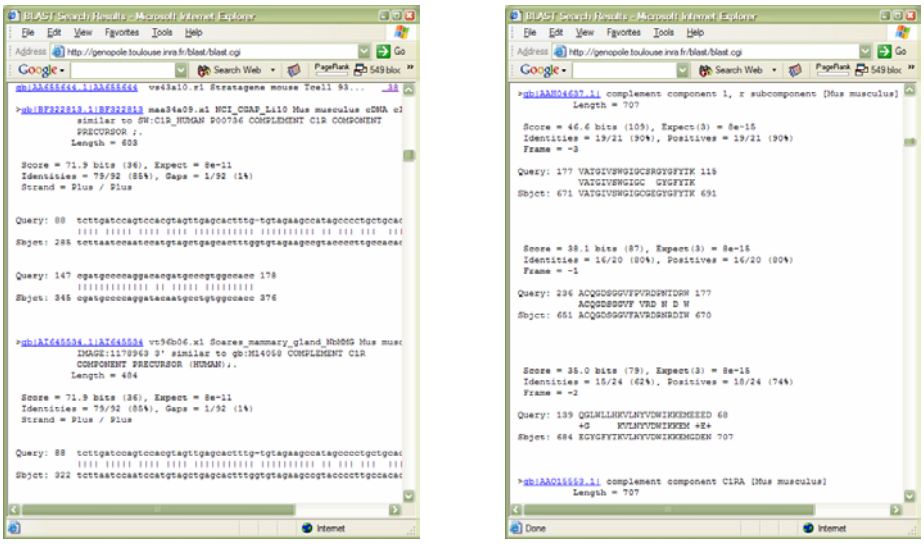
```
Let S ← source with forms f ∈ F, f = (P,B)
Let D ← the service class description with examples e ∈ E

for all f = (P,B) ∈ F do
   for all e ∈ E do
         for all a = (r,t,v) s.t. required(a) = true do
               if ¬∃p = (t,i,v) ∈ P s.t. a_t = p_t then
                     F = F – f
if F = Ø then
   processForms(F)
```

Module Selection. The modular design of the service class description framework and the DYNABOT discovery and analysis system allows many of the system components to be extended or replaced with expansion modules. For example, a service class description may reference an alternate type system or a different querying strategy than the included versions. Step 3 in the service analysis process resolves any external references that may be defined in the service class description or configuration files and loads the appropriate code components.

Query Generation. The heart of the service analysis process is the query generation, probing, and matching loop shown in steps 4, 5, and 6 of Figure 6. Generating high quality queries is a critical component of the service analysis process, as low-quality queries will result in incorrect classifications and increased processing overhead. DYNABOT's query generation component is directed by the service class description to ensure relevance of the queries to the service class. Queries are produced by matching the probing templates from the service class description with the form parameters in the source's forms; Figure 1 shows a fragment of the probing template for the nucleotide BLAST service class description.

Probing and Matching. Once the queries have been generated, the service analyzer proceeds by selecting a query, sending it to the target source, and checking the response against the result type specified in the service class description. This process is repeated until a successful match is made or the set of query probes is exhausted. On a match, the service analyzer produces a source capability profile of the target source, including the steps needed to produce a successful query.

Figure 8 shows the probing results from two different services analyzed with the same nucleotide BLAST service class description. Source (a) is a member of the nucleotide BLAST service class while source (b) is a member of the protein BLAST service class, a related type of service that uses a similar interface to nucleotide BLAST but performs a different function. Using the type definitions from the service class description, the service analyzer is able to determine that (a) is an appropriate response for a member of the nucleotide BLAST service class while (b), although structurally similar, is not an appropriate response. This information allows the service analyzer to correctly classify these two sources despite the similarity of their responses: source (a) is declared a match while source (b) is not.



(a) (b)

Figure 8: Example matching (a) and nonmatching (b) search results

Algorithm 2 presents a sketch of the query probing and matching process. Our prototype implementation includes invalid query filtering and some heuristic optimizations that are omitted from the algorithm presented here for clarity's sake. These optimizations utilize the hints specified in the probing template section of the service class description to match probing arguments with the most likely candidate form parameter. For instance, the nucleotide BLAST service class description specifies that form parameters that accept text input and are named "sequence" are very likely to be the appropriate parameter for the `DNASequence` probe argument. These hints are static and must be selected by the service class description author; our ongoing research includes a study of the effectiveness of learning techniques for matching template arguments to the correct form parameters. We expect that the system should be able to deduce a set of analysis hints from successfully matched sources which can then be used to enhance the query selection process.

Algorithm 2: Query Probing

Let S ← source with forms $f \in F$, $f = (P,B)$
Let D ← the service class description with examples $e \in E$

for all $f = (P,B) \in F$ **do**
  Let $Q \leftarrow E \times P$
    for all $q \in Q$ **do**
      Let r ← executeQuery(q)
      if *responseMatches(r,D)* **then**
        processMatch(r,q,D)

EXPERIMENTAL RESULTS

We have developed an initial set of experiments based on the DYNABOT prototype service discovery system to test the validity of our approach. The experiments were designed to test the accuracy and efficiency of DYNABOT and the service probing and matching techniques. We have divided our tests into three experiments. The first experiment is designed to test only the probing and matching components of the crawler without the confounding influence of an actual web crawl. Experiment 2 tests the performance of the entire DYNABOT system by performing a web crawl and analyzing the potential services it encounters. Experiment 3 shows the effectiveness of pruning the search space of possible services by comparing an undirected crawler with one using a more focused methodology.

The DYNABOT prototype is implemented in Java and can examine a set of supplied URLs or crawl the Web looking for sources matching a supplied service class description. All experiments were executed on a Sun Enterprise 420R server with four 450 MHz UltraSPARC-II processors and 4 GB memory. The server runs SunOS 5.8 and the Solaris Java virtual machine version 1.4.1.

Crawler Configuration. The DYNABOT configuration for these experiments utilized several modular components to vary the conditions for each test. All of the configurations used the same network interaction subsystem, in which domain name resolution, document retrieval, and form submission are handled by the HttpUnit user agent library (Gold, 2003). The experiments utilized the *service analyzer* document processing module for service probing and matching. Service analysis employed the same static service class description in all the tests, fragments of which have been shown in Figure 1 and Figure 4. All of the configurations also included the *trace generator* module which records statistics about the crawl, including URL retrieval order, server response codes, document download time, and content length. 32 crawling threads were used in each run.

We utilized two configuration variations in these experiments: the trace configuration and the random walk configuration. The trace configuration is designed to follow a predetermined path across the Web and utilizes the trace URL frontier implementation to achieve this goal. This frontier accepts a seed list in which any URLs found are crawled in the order that they appear in the list. These seed lists can be either hand generated or generated from previous crawls using the trace generator. In the trace configuration, no URLs can be added to the frontier and no attempt is made to prevent the crawler from retrieving the same URL multiple times.

The random walk configuration mimics more traditional web crawlers but attempts to minimize the load directed at any one server. In this configuration, the *link extractor* module was employed to extract hyperlinks from retrieved documents and insert them into the URL frontier. The random walk frontier implementation uses an in-memory data structure to hold the list of URLs that have yet to be crawled, from which it selects one at random when a new URL is requested. This configuration also includes a visited list, which stores hash codes of URLs that have been visited which the crawler can check to avoid reacquiring documents that have already been seen.

Experiment 1: BLAST Classification

The first experiment tested the service analyzer processing module only and demonstrates its effectiveness quantitatively, providing a benchmark for analyzing the result of our subsequent

experiments. In order to test the service analyzer, the crawler was configured to utilize the trace frontier with a hand-selected seed.

The data for this experiment consists of a list of 74 URLs that provide a nucleotide BLAST gene database search interface; this collection of URLs was gathered from the results of several manual Web searches. The sites vary widely in complexity: some have forms with fewer than 5 input parameters, while others have many form parameters that allow minute control over many of the options of the BLAST algorithm. Some of the sources include an intermediate step, called an indirection, in the query submission process. A significant minority of the sources use JavaScript to validate user input or modify parameters based on other choices in the form. Despite the wide variety of styles found in these sources, the DYNABOT service analyzer is able to recognize a large number of the sites using a nucleotide BLAST service class description of approximately 150 lines.

Crawl Statistics	
Number of matching sources	74
Total number of forms	79
Total number of form parameters	913
Total of forms submitted	1456
Maximum submissions per form	60
Average submissions per form	18.43
Number of matched sources	53
Success rate	**72.97%**

Aggregate Probe Times	
Minimum probe time	3 ms
Minimum fail time (post FormFilter)	189 s
Maximum fail time (post FormFilter)	11823 s
Average fail time (post FormFilter)	2807 s
Minimum match time (post FormFilter)	2.3 s
Maximum match time (post FormFilter)	2713 s
Average match time (post FormFilter)	284 s

Number of Probes	**Frequency**
0	12
1-10	46
11-20	1
21-30	2
31-40	2
41-50	1
51-60	10

Probe time (sec.)	**Frequency**
<0.5	3
0.5-1	1
1-5	11
5-10	5
10-50	10
50-100	2
100-500	31
>500	11

Table 2: Sites classified using the nucleotide BLAST service class description

Table 2 shows the results of Experiment 1. Sites listed as successes are those that can be correctly queried by the analyzer to produce an appropriate result, either a set of alignments or an empty BLAST result. An empty result indicates that the site was queried correctly but did not contain any results for the input query used. Since all of the URLs in this experiment were manually verified to be operational members of the service class, a perfect classifier would have achieved a success rate of 100%; the left half of Table 2 demonstrates that the DYNABOT service analyzer achieves an overall success rate of 73%.

There are several other interesting data characteristics and experimental results presented in Table 2. The relatively low number of forms per source – 79 forms for 74 sources – indicates that most of these sources use single-form entry pages. However, the average number of parameters per form is over 11 (913 parameters / 79 forms = 11.56), indicating that these forms are fairly complex. We are currently exploring form complexity analysis and comparison to determine the extent to which the structure of a source's forms can be used to estimate the likelihood that the source matches a service class description.

Source form complexity directly impacts the query probing component of the service analyzer, including the time and number of queries needed to recognize a source. To grasp the scaling problem with respect to the number of form parameters and the complexity of the service class description, consider a Web source with a single form *f* containing 20 parameters, that is $|P| = 20$. Further suppose that the service class description being used to analyze the source contains a single probing template with two arguments, $|A| = 2$, and that all of the arguments are required. The number of combinations of arguments with parameters is then $|P|$ *choose* $|A| = 190$, a large but perhaps manageable number of queries to send to a source. The number of combinations quickly spirals out of control as more example arguments are added, however: with a three-argument example the number of combinations is 1140, four arguments yields 4845, and testing a five argument example would require 15,504 potential combinations to be examined!

Despite the scalability concerns, Table 2 demonstrates the effectiveness of the SCD-directed probing strategy: most of the sources were classified with less than 10 probes (58) in less than 500 seconds (63). These results indicate the effectiveness of the static optimizations employed by the service analyzer such as the probing template hints. Our ongoing research includes an investigation of the use of learning techniques and more sophisticated query scoring and ranking to reduce these requirements further and improve the efficiency of the service analyzer.

Failed sites are all false negatives that fall into two categories: indirection sources and processing failures. An indirection source is one that interposes some form of intermediate step between entering the query and receiving the result summary. For example, NCBI's BLAST server contains a formatting page after the query entry page that allows a user to tune the results of their query. Simpler indirection mechanisms include intermediate pages that contain hyperlinks to the results. We do not consider server-side or client-side redirection to fall into this category as these mechanisms are standardized and are handled automatically by Web user agents. Recognizing and moving past indirection pages presents several interesting challenges because of their free-form nature. Incorporating a general solution to complex, multi-step Web sources is part of our ongoing work (Ngu et al., 2003).

Processing errors indicate problems emulating the behavior of standard Web browsers. For example, some Web design idioms, such as providing results in a new window or multi-frame interfaces, are not yet handled by the prototype. Support for sources that employ JavaScript is also incomplete. We are working to make our implementation more compliant with standard Web browser behavior. The main challenge in dealing with processing failures is accounting for them in a way that is generic and does not unnecessarily tie site analysis to the implementation details of particular sources.

Experiment 2: BLAST Crawl

Our second experiment tested the performance characteristics of the entire DYNABOT crawling, probing, and matching system. The main purpose of this experiment is to demonstrate the need for a directed approach to service discovery. Intuitively, the problem stems from the characteristics of the service Web environment: instances of a particular service class, such as nucleotide BLAST, will make up a small fraction of the sites related to the relevant domain, e.g. bioinformatics. Likewise, the sites belonging to any particular domain will constitute a small portion of the complete Web. Experiment 2 provides evidence to support this conjecture and demonstrates the need for intelligent service discovery and resource allocation. An effective service discovery mechanism must use its resources wisely by spending available processing power on sources that are more likely to belong to the target set.

Crawl Statistics	
Number of URLs crawled	1349
Number of sites with forms	467
Total number of forms	686
Total number of form parameters	2837
Total of forms submitted	4032
Maximum submissions per form	10
Average submissions per form	5.88
Number of matched sources	**2**

Response Code	Frequency
200	1212
30x	114
404	18
50x	6

Content Type	Frequency
text/html	1238
application/pdf	36
text/plain	23
other	52

Table 3: Results from 06022004 crawl, Google 100 BLAST seed, random walk URL frontier

The results of this experiment are presented in Table 3. For this test, the crawler was configured utilizing the random walk URL frontier with link extraction and service analysis. The initial seed for the frontier was the URLs contained in the first 100 results returned by Google for the search "bioinformatics BLAST." URLs were returned from the frontier at random and all retrieved pages had their links inserted into the frontier before the next document was retrieved. These results are not representative of the Web as a whole, but rather provide insight into the characteristics of the environment encountered by the DYNABOT crawler during a domain-focused crawl. The most important feature of these results is the relatively small number of matched sources: despite the high relevance of the seed and subsequently discovered URLs to the search domain, only a small fraction of the services encountered matched the service class description. The results from Experiment 1 demonstrate that the success rate of the service analyzer is very high, leading us to believe that the nucleotide BLAST services make up only a small percentage of the bioinformatics sites on the Web. This discovery does not run counter to our intuition; rather, it suggests that successful and efficient discovery of domain-related services hinges on the ability of the discovery agent to reduce the search space by pruning out candidates that are unlikely to match the service class description.

Experiment 3: Directed Discovery

Given the small number of relevant Web services related to our service class description, Experiment 3 further demonstrates the effectiveness of pruning the discovery search space in order to find high quality candidates for probing and matching. One important mechanism for document pruning is the ability to recognize documents and links that are relevant or point to relevant sources before invoking the expensive probing and matching algorithms. Using the random walk crawler configuration as a control, this experiment tests the effectiveness of using link hints to guide the crawler toward more relevant sources. The link hint frontier is a priority-based depth-first exploration mechanism in which hyperlinks that match the frontier's hint list are explored before nonmatching URLs. For this experiment, we employed a static hint list using a simple string containment test for the keyword "blast" in the URL.

Table 4 presents the results. The seed lists for the URL frontiers in this experiment were similar to those used in Experiment 2 except that 500 Google results were retrieved and all the Google cache links were removed. The link hint focused crawler discovered and matched 15 Web sources with a fewer number of trials per form then its random walk counterpart. Although the

number of URLs crawled in the both tests was roughly equivalent, the link hint crawler found sources of much higher complexity as indicated by the total number of form parameters found: 1038 for the link hint crawler versus 348 for the random walk crawler.

Crawl Statistics	
Number of URLs crawler	174
Number of sites with forms	74
Total number of forms	108
Total number of form parameters	348
Total of forms submitted	2996
Maximum submissions per form	60
Average submissions per form	27.74
Number of matched sources	**0**

(a) Random walk URL frontier

Crawl Statistics	
Number of URLs crawler	182
Number of sites with forms	71
Total number of forms	137
Total number of form parameters	1038
Total of forms submitted	3340
Maximum submissions per form	60
Average submissions per form	24.38
Number of matched sources	**15**

(b) LinkHint "blast" frontier

Table 4: Results from 06022004 crawl, Google 500 BLAST seed

The results of Experiment 3 suggest a simple mechanism for selecting links from the URL frontier to move the crawler toward high quality candidate sources quickly: given a hint word, say "blast," first evaluate all URLs that contain the hint word, proceeding to evaluate URLs that do not contain the hint word only after the others have been exhausted. This scheme can be quite easily implemented using a priority queue. However, the hint list is static and must be selected manually. We are investigating the effectiveness of learning algorithms and URL ranking algorithms for URL selection. This URL selection system would utilize a feedback loop in which the "words" contained in URLs would be used to prioritize the extraction of URLs from the frontier. Words contained in URLs that produced service class matches would increase the priority of any URLs in the frontier that contained those words, while words that appeared in nonmatching URLs would likewise decrease their priority. In order to be effective, this learning mechanism would also need a word discrimination component, such as term frequency inverse document frequency (TFIDF) measure, so that common words like "http" would have little effect on the URL scoring.

Discussion

The results of our experiments demonstrate the effectiveness of the service class model and the DYNABOT discovery and matching agent. The results also suggest areas for further exploration to optimize the search and analysis process.

We are exploring the potential of dynamic learning techniques for reducing the resource consumption of the service analyzer by limiting the amount of effort it expends analyzing unlikely services. These techniques would perform one or more of the following functions: service filtering, maximum probe count adjustment, and query probe reordering. In *service filtering*, the analyzer would evaluate the service based on its forms and eliminate it from consideration or reprioritize it if the service is unlikely to be a service class match. *Maximum probe count adjustment* would allow the service analyzer to dynamically adjust the number of queries attempted on a per-source basis using a comparison with previously encountered services. *Query probe reordering* would allow the service analyzer to dynamically reorder query probes like the static reordering described previously but using information gathered dynamically during the crawl.

The current DYNABOT prototype includes one service filtering optimization, form filter,

which eliminates any pages from consideration that either contain no form elements or whose form elements do not match the domain as defined by the service class description. For instance, a page containing a form with only list boxes and radio buttons would be eliminated from consideration if the service class description specified a free text input. This facility could be expanded to utilize information gathered from previous probing and matching operations. Using form similarity comparison, the service analyzer could measure the forms in a candidate service against previously matched sources and, based on this comparison, eliminate the service from consideration or adjust the probe count and order.

Another optimization that can be used to guide the analysis process is document text analysis. Many services contain technical jargon or other specialized vocabularies that distinguish these documents from those in other domains. Using techniques like the Levenshtein string edit distance (Levenshtein, 1966) and term frequency analysis could help direct the crawler toward relevant hubs but might also be useful when performed on the start pages of services themselves. Much like form similarity comparison, these techniques could be used to adjust the service analysis properties based on the result of the comparison: services that are likely to match would be allocated more resources than those that are not.

These techniques should improve the recognition accuracy of the DYNABOT system, allowing the crawler to recognize difficult services and expanding the possible types of sources amenable to discovery and classification using the DYNABOT service discovery approach. Our ongoing research efforts include the development of several of these optimizations for the DYNABOT prototype; part of this development effort includes exploring other service classes that would provide meaningful test data and demonstrate DYNABOT's applicability to a wide range of service types.

RELATED WORK

Web crawlers have been searching and indexing the static Web since nearly the time of its creation. Starting from a set of seed pages, a crawler traverses the Web and processes the sites it encounters while extracting new hyperlinks to crawl from the encountered sites. Crawlers have generated commercial and research interest due to their popularity (Pew, 2002) and the technical challenges involved with scaling a crawler to handle the entire Web (Brin & Page, 1998; Miller & Bharat, 1998; Heydon & Najork, 1999; Broder et al., 2003). There is active research into topic driven or focused crawlers (Chakrabarti et al., 1999) which crawl the Web looking for sites relevant to a particular topic; Srinivasan et al. (2002) present such a crawler for biomedical sources that includes a treatment of related systems.

Our research seeks to unify complex Web data sources using automatic discovery and capability detection; the BLAST family of data sources have provided a test case for our approach (Rocco & Critchlow, 2003; Ngu et al., 2003). The ShopBot agent (Doorenbos et al., 1997) uses a similar approach and is designed to assist users in the task of online shopping. ShopBot uses a domain description that lists useful attributes about the services in question. The authors addressed the problems of learning unknown vendor sites and integrating a set of learned sources into a single interface. Our present work addresses the related problem of automatically classifying services from an arbitrary set of sites. The service class description format we describe provides greater descriptive power than ShopBot's domain descriptions and can specify complex data types and source control flow information.

Researchers have also examined heterogeneous data integration in the domain of biological data. DiscoveryLink (Haas et al., 2001) provides access to wrapped data sources and includes query planning and optimization capabilities. Eckman et al. (2001) present a similar system with a comparison to many existing related efforts.

CONCLUSION

We have presented DYNABOT, a crawler designed to discover and analyze dynamic Web data sources relevant to a domain of interest. DYNABOT's use of the service class model of the Web, through the construction of service class descriptions, allows an abstract rendition of the target domain to guide the crawler toward relevant sources and probe them for their capabilities. DYNABOT employs a modular, self-tuning crawling architecture and algorithms for efficient probing of the Deep Web. Our experimental results demonstrate the effectiveness of the service class discovery mechanism which achieves recognition rates of up to 73%. These results offer effective techniques for efficiently managing service discovery in the face of the immense scale of the Deep Web.

ACKNOWLEDGMENT

This work is performed under a subcontract from LLNL under the LDRD project. The work of the first three authors is also partially supported by the National Science Foundation under a CNS Grant, an ITR grant, and a DoE SciDAC grant, an IBM SUR grant, an IBM faculty award, and an HP equipment grant. The work of the fourth author is performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract No.W-7405-ENG-48. Our thanks are also due to those past team members who have contributed to the project in one way or the other, especially David Buttler, Wei Han, and Jianjun Zhang.

REFERENCES

Bergman, M.K. (2003). The DeepWeb: Surfacing Hidden Value. http://www.completeplanet.com/Tutorials/DeepWeb/, 2003.

Brin, S. & Page, L. (1998). The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems*, 30(1–7):107–117, 1998.

Broder, A., Najork, M., & Weiner, J. (2003). Efficient URL caching for worldwide web crawling. In *Proceedings of the International World Wide Web Conference*, 2003.

Chakrabarti, S., Berg, M. van den, & Dom, B. (1999). Focused crawling: A new approach to topic-specific web resource discovery. In *Proceedings of the International World Wide Web Conference*, 1999.

Chang, K., He, B., Li, C., Patel, M., & Zhang, Z. (2004). Structured Databases on the Web: Observations and Implications. *SIGMOD Record*, 33(3): 61-70, 2004.

Doorenbos, R. B., Etzioni, O., & Weld, D. S. (1997). A scalable comparison-shopping agent for the world-wide web. In W. L. Johnson and B. Hayes-Roth, editors, *Proceedings of the First International Conference on Autonomous Agents (Agents'97)*, pages 39–48, Marina del Rey, CA, USA, 1997.

Eckman, B., Lacroix, Z., & Raschid, L. (2001). Optimized seamless integration of biomolecular data. In *IEEE International Conference on Bioinformatics and Biomedical Engineering*, pages 23–32, 2001.

Fallside, D.C. (2001). XML Schema Part 0: Primer. *Technical report, World Wide Web Consortium*, http://www.w3.org/TR/xmlschema-0/, 2001.

Gold, R. (2003). HttpUnit. http://httpunit.sourceforge.net, 2003.

Haas, L., Schwarz, P., Kodali, P., Kotlar, E., Rice, J., & Swope, W. (2001). Discoverylink: A system for integrating life sciences data. *IBM Systems Journal*, 40(2), 2001.

Heydon, A. & Najork, M. (1999). Mercator: A scalable, extensible web crawler. *World Wide Web*, 2(4):219–229, 1999.

Lawrence, S. & Giles, C.L. (1998). Searching the world wide web. *Science*, 280(5360):98, 1998.

Lawrence, S. & Giles, C. L. (1999). Accessibility of information on the web. *Nature*, 400:107–109, 1999.

Levenshtein, V. I. (1966). Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, 10:707–710, 1966.

Miller, R. & Bharat, K. (1998). SPHINX: A framework for creating personal, site-specific web crawlers. In *Proceedings of the International World Wide Web Conference*, 1998.

Ngu, A. H. H., Rocco, D., Critchlow, T., & Buttler, D. (2003). Automatic discovery and inferencing of complex bioinformatics web interfaces. *Technical Report UCRL-JRNL-201611, Lawrence Livermore National Laboratory*, 2003.

Pew Internet and American Life Project Survey (2002). Search engines: a Pew Internet project data memo. http://www.pewinternet.org/reports/toc.asp?Report=64, July 2002.

Rocco, D. & Critchlow, T. (2003). Automatic Discovery and Classification of Bioinformatics Web Sources. *Bioinformatics*, 19(15):1927–1933, 2003.

Srinivasan, P., Mitchell, J., Bodenreider, O., Pant, G., & Menczer, F. (2002). Web crawling agents for retrieving biomedical information. In *Proceedings of the International Workshop on Agents in Bioinformatics (NETTAB-02)*, 2002.

ABOUT THE AUTHORS

Daniel Rocco is an Assistant Professor at the University of West Georgia's Department of Computer Science. His primary responsibility is teaching at both the undergraduate and graduate level: he has taught courses including introductory CS, senior capstone, Web technologies, and database systems. He has also been responsible for developing and piloting several new course offerings including an introductory media and gaming class. Dr. Rocco's other responsibilities include research, curriculum development, and a variety of services to the institution. Before joining the faculty at UWG, Dr. Rocco was a member of the DiSL research lab at Georgia Institute of Technology's College of Computing, where he earned his PhD in 2004. His research interests include Web technologies, database systems, voice recognition interfaces, and computer science education.

James Caverlee is currently a PhD student in the College of Computing at the Georgia Institute of Technology. His research interests are focused on Web information management and retrieval, Web services, and Web based data intensive systems and applications. James graduated magna cum laude from Duke University in 1996 with a BA in Economics. He received the MS degree in Engineering-Economic Systems & Operations Research in 2000 and the MS degree in Computer Science in 2001, both from Stanford University. His PhD dissertation research is focused on Web Information Retrieval and Web data mining, including efficient and spam-resilient Web search algorithms.

Ling Liu is currently an associate professor at the College of Computing at Georgia Tech. There, she directs the research programs in Distributed Data Intensive Systems, examining research issues and technical challenges in building large scale distributed computing systems that can grow without limits. Dr. Liu and the DiSL research group have been working on various aspects of distributed data intensive systems, ranging from decentralized overlay networks, exemplified by peer to peer computing and grid computing, to mobile computing systems and location based services, sensor network systems, and enterprise computing technology. She has published over 100 international journal and conference articles. Her research group has produced a number of software systems that are either open sources or directly accessible online, among which the most popular ones are WebCQ and XWRAPElite. She and her students have received a best paper award from IEEE ICDCS 2003 for their work on PeerCQ and a best paper award from International Conference of World Wide Web (2004) for their work on Caching Dynamic Web Content. Most of Dr. Liu's current research projects are sponsored by NSF, DoE, DARPA, IBM, and HP. She was a recipient of IBM Faculty Award (2003) and a participant of the Yamacraw program, State of Georgia.

Terence Critchlow is the team lead for the BioEncyclopedia effort within the Biodefense Knowledge Center (BKC) at the Lawrence Livermore National Lab (LLNL). In this capacity, he is leading a large team of researchers and developers in an effort to integrate relevant biodefense information into a consistent environment for use by BKC analysts. Prior to working with the BKC, Dr. Critchlow led several research projects focusing on improving scientists' interactions with large data sets. He obtained a PhD in Computer Science from the University of Utah in 1997 and has been a member of the Center for Applied Scientific Computing at LLNL ever since.