Domain-specific Web Service Discovery with Service Class Descriptions*

Daniel Rocco
University of West Georgia
Carrollton, GA, USA
drocco@westga.edu

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

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

Terence Critchlow
Lawrence Livermore National Laboratory
Livermore, CA, USA
critchlow1@llnl.gov

Abstract

This paper presents DynaBot, a domain-specific web service discovery system. The core idea of the DynaBot service discovery system is to use domain-specific service class descriptions powered by an intelligent Deep Web crawler. In contrast to current registry-based service discovery systems – like the several available UDDI registries – DynaBot promotes focused crawling of the Deep Web of services and discovers candidate services that are relevant to the domain of interest. It uses intelligent filtering algorithms to match services found by focused crawling with the domain-specific service class descriptions. We demonstrate the capability of DynaBot through the BLAST scenario and describe our initial experience with DynaBot.

1 Introduction

With the increasing adoption of web services and the Service-Oriented Computing paradigm [13], there is a growing need for efficient and effective mechanisms for web service discovery. Web service discovery is critical in a number of contexts, from helping organizations deploy flexible, re-configurable architectures to identifying appropriate service partners and competitors, and so on.

Current web service discovery techniques typically rely on registry-based discovery. In the registry-based approach, services advertise their existence and capabilities with a service registry like the ones offered by Microsoft [`http://uddi.microsoft.com`] and IBM [`http://uddi.ibm.com`]. Interested users may discover relevant services by querying the metadata maintained in the registry or by browsing the registry. However, registry-based discovery systems have several drawbacks. Many of these technologies are still evolving and have limited deployment. In addition, registry-based discovery relies on services correctly advertising themselves in a known repository, effectively limiting the number of services that can be discovered. Finally, the limited descriptive power in existing registry standards implies that service analysis is still required to ascertain a service's capabilities.

With these challenges in mind, we present DYNABOT, a domain-specific web service discovery system for effectively identifying domain-specific web services. DYNABOT is a complementary approach to traditional registry-based discovery. The core idea of the Dynabot service discovery system is to use domain-specific service class descriptions powered by an intelligent Deep Web crawler. In contrast to current registry-based service discovery systems, Dynabot promotes focused crawling of the Deep Web of services and discovers candidate services that are relevant to the domain of interest. It uses intelligent filtering algorithms to match services found by focused crawling with the domain-specific service class descriptions. The major challenges facing DYNABOT are identification of potential web services, classifying the discovered services, managing data generated throughout the classification process, and ranking of both services and the results they produce. DYNABOT uses its service class model with associated service class descriptions to determine the capabilities of discovered services and to classify web services as members of a service class.

We demonstrate the capability of Dynabot through the BLAST 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.

2 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

*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.

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 $s_1, ..., s_k$ ($k \geq 1$). 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: $SCD =< \mathcal{T}, \mathcal{G}, \mathcal{P} >$, where $\mathcal{T}$ denotes a set of *type definitions*, $\mathcal{G}$ denotes a *control flow graph*, and $\mathcal{P}$ 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 [**B**asic **L**ocal **A**lignment **S**earch **T**ool] 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.

```
<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.

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.

2.1 Type Definitions

The first component of a service class description specifies the data types $t \in \mathcal{T}$ 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 [6] 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.

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 may be composed into *complex types*.

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

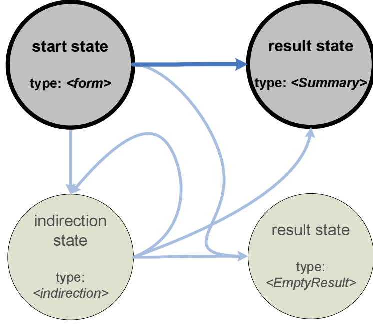

Figure 2: Nucleotide BLAST: control flow graph.

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.

2.2 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 $\mathcal{G} = (E, V)$, 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 $t \in \mathcal{T}$. 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 Section 3.2, 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.

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.

2.3 Probing Templates

The third component of the service class description is the set of probing templates $\mathcal{P}$, 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. 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 3 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

```
<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 3: Nucleotide BLAST: probing template.

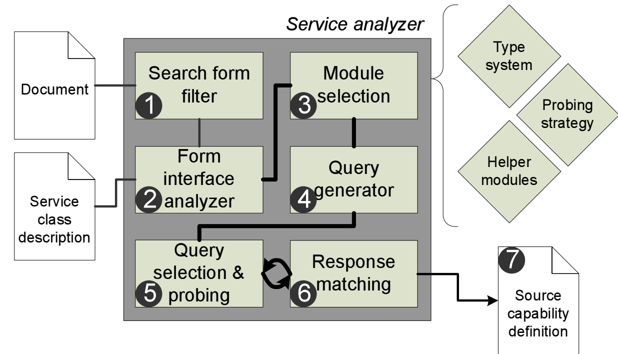

Figure 4: DYNABOT Service Analyzer

type section of the SCD.

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."

3 DYNABOT 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, registry-based discovery systems 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.

3.1 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 [5]. 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 [8], 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.

3.2 Service Analyzer

The process of web service 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 candidate service. The *service analyzer* consists of a form filter and analyzer, an extension mechanism, a query generator, a query prober, and a response matcher.

Overview. When the processor encounters a new service site to test, its first task is to invoke the *form filter*, which ensures that the candidate service has a form interface (Figure 4(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 de-

scription, the service has matched the service class description and a source capability profile (7) is produced as the output of the analysis process. The profile contains the specific steps needed to successfully query the web service. If the probe was unsuccessful, additional probing queries can be attempted.

Definitions. The process of analyzing a service begins when the crawler passes a potential URL for evaluation to the service analysis processing module. A service provider S consists of an initial set of forms F, each of which corresponds to a candidate service s. 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 process of *query probing* involves manipulating a service provider's forms to ascertain their purpose with the ultimate goal of determining the function of the service itself. Although the expected inputs and purpose of each of the various parameters and forms 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 service provider'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.

Form Filter and Analyzer. The *form filter* processing step helps to reduce the service search space by eliminating any candidate service that cannot possibly match the current service class description. In the filtration step, shown in step 1 of Figure 4, the form filter eliminates any service provider S from consideration if the its form set is empty, that is $F = \emptyset$. In form analysis, shown in step 2, the service class description will be compared with the service provider, allowing the service analyzer to eliminate any forms that are incompatible with the service class description.

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 4. 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 service provider's forms; Figure 3 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 service, 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 service, including the steps needed to produce a successful query.

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 named "sequence" that accept text input 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 parameters. We expect that the system should be able to deduce a set of analysis hints from successfully matched services which can then be used to enhance the query selection process.

4 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 services 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 VM 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

Table 1: Services classified using the Nucleotide BLAST SCD

Crawl Statistics	
Number of BLAST services analyzed	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 services	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

resolution, document retrieval, and form submission are handled by the HttpUnit user agent library [7]. 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 Figures 1 and 3. 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.

4.1 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 results of our subsequent experiments. The crawler was configured to utilize the trace frontier with a hand-selected seed list.

Table 2: Experiment 1 probing statistics.

Number of probes	**Freq**
0	12
1–10	46
11–20	1
21–30	2
31–40	2
41–50	1
51–60	10

Probe time (s)	**Freq**
<0.5	3
0.5–1	1
1–5	11
5–10	5
10–50	10
50–100	2
100–500	31
>500	11

The data for this experiment consists 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 services include an intermediate step, called an indirection, in the query submission process. A significant minority of the services 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 services, the DynaBot service analyzer is able to recognize a large number of the sites using a `Nucleotide BLAST` SCD of approximately 150 lines.

Tables 1 and 2 show 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 achieved a success rate of 100%; Table 1 demonstrates that the DynaBot service analyzer achieves an overall success rate of 73%.

There are several other notable results in the data presented in Tables 1 and 2. The relatively low number of forms per service—79 forms for 74 services—indicates that most of these services 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 service's forms can be used to estimate the likelihood that the service matches a service class description.

Form complexity directly impacts the query probing component of the service analyzer, including the time and number of queries needed to recognize a service. To grasp the scaling problem with respect to the number of form parameters and the complexity of the service class description, consider a web service with a single form f containing 20 parameters, that is $|P| = 20$. Further suppose that the service class description being used to analyze the service contains a single probing tem-

Table 3: Crawl statistics from 6/2/2004 crawl, random walk frontier.

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 services	**2**

plate with two arguments, $|A| = 2$, and that all of the arguments are required. The number of combinations of arguments with parameters is then $\binom{|P|}{|A|} = \binom{20}{2} = 190$, a large but perhaps manageable number of queries to send to a service. The number of combinations quickly increases as more example arguments are added, however: with a three-argument example the number of combinations is 1140, four arguments yields 4845, and five arguments yields 15,504 potential combinations.

Despite the scalability concerns, Table 2 demonstrates the effectiveness of the SCD-directed probing strategy: most of the services were classified with less than 10 probes (58) in less than 500 seconds (63). This indicates 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 service analyzer efficiency.

Failed sites (27%) are all false negatives that fall into two categories: indirection services and processing failures. An indirection service is one that interposes some form of intermediate step between entering the query and receiving the result summary. For example, NCBI's [12] 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 services is part of our ongoing work [11].

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 sites 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 services.

4.2 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. Since instances of a particular service class, such as `Nucleotide BLAST`, will constitute a small portion of the complete Web, an effective service discovery mechanism must use its resources wisely by spending available processing power on services that are more likely to belong to the target set.

Table 4: Response types from 6/2/2004 crawl, random walk frontier.

Response	**Freq**	**Content Type**	**Freq**
200	1212	text/html	1238
30x	114	application/pdf	36
404	18	text/plain	23
50x	6	other	52

The results of this experiment are presented in Tables 3 and 4. 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 services: 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.

4.3 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 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 services 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 services. 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 [Table 5], we employed a static hint list using a simple

Table 5: Results from 6/2/2004 crawl, Google 500 BLAST seed.

Crawl Statistics	Random Walk	LinkHint "blast"
Number of URLs crawled	174	182
Number of sites with forms	74	71
Total number of forms	108	137
Total number of form parameters	348	1038
Total of forms submitted	2996	3340
Maximum submissions per form	60	60
Average submissions per form	27.74	24.38
Number of matched services	**0**	**12**

string containment test for the keyword "blast" in the URL.

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 12 services with a fewer number of trials per form than its random walk counterpart. Although the number of URLs crawled in the both tests was roughly equivalent, the link hint crawler found services 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.

The results of Experiment 3 suggest a simple mechanism for selecting links from the URL frontier to move the crawler toward high quality candidate services 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. Since the hint list is static and must be selected manually, we are studying 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. This learning mechanism would also need a word discrimination component (such as TFIDF) so that common words like "http" would have little effect on the URL scoring.

5 Related Work

Researchers have previously explored different aspects of the service discovery problem, ranging from discovery in a federated environment [16], to identifying services that meet certain quality-of-service guarantees [9], to evaluating services based on a distributed reputation metric [17], to other quality metrics like in [19]. A personalized approach to service discovery was previously presented in [3].

Web crawlers have generated commercial and research interest due to their popularity and the technical challenges involved with scaling a crawler to handle the entire Web [1, 10, 8, 2]. There is active research into topic-driven or focused crawlers [4] which crawl the Web looking for sites relevant to a particular topic; Srinivasan et al. [18] present such a crawler for biomedical services that includes a treatment of related systems. A previous effort to crawl the "Hidden" Web suggested using domain-specific vocabularies for filling out web forms [14]. In previous work, we have discussed some of the principles underlying DYNABOT [15].

6 Conclusion

We have presented DYNABOT, a domain-specific web service discovery system for effectively identifying domain-specific web services. The core idea of the DynaBot service discovery system is to use domain-specific service class descriptions powered by an intelligent Deep Web crawler. In contrast to current registry-based service discovery systems, DynaBot promotes focused crawling of the Deep Web of services and discovers candidate services that are relevant to the domain of interest. 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 growing number of web services.

References

[1] S. Brin and L. Page. The anatomy of a large-scale hypertextual Web search engine. In *WWW '98*.

[2] A. Broder, M. Najork, and J. Weiner. Efficient url caching for worldwide web crawling. In *WWW '03*.

[3] J. Caverlee, L. Liu, and D. Rocco. Discovering and ranking web services with BASIL: A personalized approach with biased focus. In *2nd Int'l Conference on Service-Oriented Computing*, 2004.

[4] S. Chakrabari et al. Focused crawling: A new approach to topic-specific web resource discovery. In *WWW '99*.

[5] K. C.-C. Chang et al. Structured databases on the web: Observations and implications. *SIGMOD Record*, 33(3), Sept 2004.

[6] D. C. Fallside. XML Schema Part 0: Primer, 2001.

[7] R. Gold. HttpUnit. http://httpunit.sourceforge.net, 2003.

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

[9] Y. Liu, A. H. Ngu, and L. Zeng. QoS computation and policing in dynamic web service selection. In *WWW '04*.

[10] R. Miller and K. Bharat. SPHINX: A framework for creating personal, site-specific web crawlers. In *WWW '98*.

[11] A. H. H. Ngu, D. Rocco, T. Critchlow, and D. Buttler. Automatic discovery and inferencing of complex bioinformatics web interfaces. *World Wide Web Journal*, 2005.

[12] NLM/NIH. National Center for Biotechnology Information. http://www.ncbi.nih.gov/.

[13] M. P. Papazoglou. Service-oriented computing: Concepts, characteristics and directions. In *WISE '03*.

[14] S. Raghavan and H. Garcia-Molina. Crawling the hidden web. In *VLDB '01*.

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

[16] K. Sivashanmugam, K. Verma, and A. Sheth. Discovery of web services in a federated registry environment. In *ICWS '04*.

[17] R. M. Sreenath and M. P. Singh. Agent-based service selection. In *Journal on Web Semantics (JWS), 2003*.

[18] P. Srinivasan et al. Web crawling agents for retrieving biomedical information. In *Workshop on Agents in Bioinformatics*, 2002.

[19] L. Zeng, B. Benatallah, M. Dumas, J. Kalagnanam, and Q. Z. Sheng. Quality driven web services composition. In *WWW '03*.