

Towards High-Precision Service Retrieval

Abraham Bernstein¹, Mark Klein²

¹NYU – Stern School of Business, 44 West 4th Street, Suite 9-76, New York, NY 10012, U. S. A.
avi@acm.org

²MIT – Center for Coordination Science, 77 Massachusetts Ave, Room NE20-336, Cambridge, MA 02139, U. S. A.
m_klein@mit.edu

Abstract. The ability to rapidly locate useful on-line services (e.g. software applications, software components, process models, or service organizations), as opposed to simply useful documents, is becoming increasingly critical in many domains. Current service retrieval technology is, however, notoriously prone to low precision. This paper describes a novel service retrieval approach based on the sophisticated use of process ontologies. Our preliminary evaluations suggest that this approach offers qualitatively higher retrieval precision than existing (keyword and table-based) approaches without sacrificing recall and computational tractability/scalability.

1 The Challenge: High Precision Service Retrieval

Increasingly, on-line repositories such as the World Wide Web are being called upon to provide access not just to documents that collect useful *information*, but also to *services* that describe or even provide useful *behavior*. Potential examples of such services abound:

- *Software applications* such as web services (e.g. for engineering, finance, meeting planning, or word processing) that can be invoked remotely by people or software. See, for example, www.salcentral.com.
- *Software components* that can be downloaded for use when creating a new application. See, for example, www.mibsoftware.com and www.compoze.com.
- *Best practice repositories* that describe how to achieve some goal. See, for example, process.mit.edu/eph/ and www.bmpcoe.com.
- *Individuals* or *organizations* who can perform particular functions, e.g. as currently brokered using such web sites as guru.com, elance.com and freeagent.com.

As the sheer number of such services increase it will become increasingly important to provide tools that allow people (and software) to quickly find the services they need, while minimizing the burden for those who wish to list their services with these search engines [1]. Current service retrieval approaches have, however, serious limitations with respect to meeting these challenges. They either perform relatively poorly or make unrealistic demands of those who wish to index or retrieve services. This paper first reviews these approaches and then presents as well as evaluates a novel service retrieval approach based on the sophisticated use of process ontologies. It closes with a discussion of related work and open challenges for future work.

2. The State of the Art

Current service retrieval technology has emerged from several communities. The information retrieval community has focused on the retrieval of natural language documents, not services per se, and has as a result emphasized keyword-based approaches. The software agents and distributed computing communities have developed simple ‘table-based’ approaches for ‘matchmaking’ between tasks and on-line services. The software engineering community has developed by far the richest set of techniques for service retrieval [2]. We can get a good idea of the relative merits of these approaches by placing them in a *precision/recall* space (Figure 1):

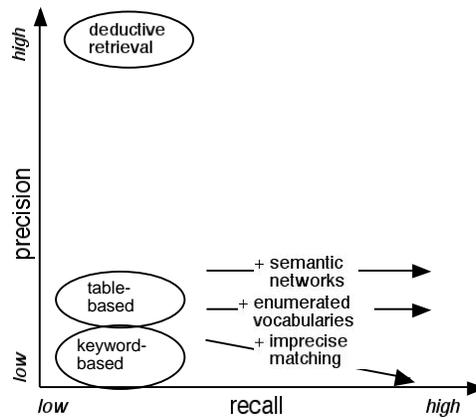


Fig. 1. The state of the art in service retrieval

Recall is the extent to which a search engine retrieves *all* of the items that one is interested in (i.e. avoiding false negatives) while *precision* is the extent to which the tool retrieves *only* the items that one is interested in (i.e. avoiding false positives) [3].

Most search engines, including those used in such service repositories as www.salcentral.com and www.uddi.org, look for items that contain the *keywords* in the query. More sophisticated variants (based on the technique known as TFIDF) prioritize matches in which the searched-for keywords are more common than usual, thereby increasing effective precision [4]. Typically, no manual effort is needed to list items with such search engines, and queries can be specified without needing to know a specialized query language. Keyword-based approaches are, however, prone to both low precision and imperfect recall. Irrelevant items may include the keywords in the query, leading to low precision. In addition, relevant items may be described using terms different from, but synonymous to, the keywords in the query, leading to imperfect recall. While techniques are available to enable acceptable levels of recall (including pre-enumerated vocabularies [5], semantic nets [6] and partial matching), keyword-based approaches remain notoriously prone to poor precision. The key underlying problem is that keywords are a poor way to capture the semantics of a query or item. If this semantics could be captured more accurately then precision would increase.

Table-based approaches [7] [8] [9] [10] [11] [12] are a second, increasingly popular, class of service model. A table-based service model consists of attribute value pairs describing the properties of an item. Figure 2 for example shows a table-based model for an integer averaging service:

| | |
|--------------------|---|
| Description | a service to find the average of a list of integers |
| Input | Integers |
| Output | Real |
| Duration | number of inputs * 0.1 msec |

Fig. 2. A table-based description of an integer sorting service

Both items and queries are described as tables: matches represent items whose property values match those in the query. All the commercial service search technologies we are aware of (e.g. Jini™, eSpeak, Salutation, UDDI/WSDL, [13]) use the table-based approach. The more sophisticated search tools emerging from the research community [14] [15] use ontologies and semantic nets to increase recall, e.g. returning a match if the input type of a service is equal to *or* a generalization of the input type specified in the query.

Table-based models, however, do little to increase precision because of the impoverished range of information they capture. These models typically include a detailed description of how to *invoke* the service (i.e., parameter types, return types, calling protocols, etc.), but don't describe what the service actually *does*, aside from an optional full-text description. The invocation-related information is of limited value for search purposes because services with different goals (e.g. services that compute averages, medians, quartiles, etc.) can share similar call signatures. Precision can be increased, however, in technologies such as UDDI that allow services to be classified using pre-defined taxonomies.

A third important class of search technique is *deductive retrieval* [16] [17] [18] wherein service and query semantics are expressed formally using logic (Figure 3):

| | |
|---------------|---|
| Name: | set-insert |
| Syntax: | set-insert(Elem, Old, New) |
| Input-types: | (Elem:Any), (Old:SET) |
| Output-types: | (New: SET) |
| Precond: | $\neg member(Elem, Old)$ |
| Postcond: | $member(Elem, New) \wedge$ $\forall x (member(x, Old) \rightarrow member(x, New)) \wedge$ $\forall y (member(y, New) \rightarrow (member(x, Old) \vee y = Elem))$ |

Fig. 3. A service description for deductive retrieval.

Retrieval consists of finding the items that can be *proven* to achieve the functionality described in the query. If we assume a non-redundant pre-enumerated vocabulary of logical predicates and a complete formalization of all relevant service and query properties, then deductive retrieval can in theory achieve both perfect precision and perfect recall. This approach, however, faces two very serious practical difficulties. First of all, it can be prohibitively difficult to model the semantics of non-trivial queries and services using formal logic. Even the simple set-insert function shown above in Figure 2 is non-trivial to formalize correctly: imagine trying to formally model the behavior of Microsoft Word or an accounting package! The second difficulty is that the proof process required can have a high computational complexity, making it extremely slow [16]. Our belief is that these limitations, especially the first one, make deductive retrieval unrealistic as a scalable general purpose service search approach.

Other approaches exist, with specialized applications. One is execution-based retrieval, wherein software components are selected by comparing their actual I/O behavior with the desired I/O behavior. This approach is suitable only for contexts where observing a few selected samples of I/O behavior are sufficient to prune the service set [19] [20] [21].

3 Our Approach: Exploiting Process Ontologies

Our challenge can thus be framed as being able to capture enough service and query semantics to substantively increase precision without reducing recall or making it unrealistically difficult for people to express these semantics. *Our central claim is that these goals can be achieved through the sophisticated use of process ontologies* [22]. In our approach, the salient behavior of a service is captured using process models, and these process models, as well as their components (subtasks, resources, and so on), are placed in the appropriate locations in the process ontology. Queries can then be defined (using a *process query language* we call PQL) to find all the services whose process models include a given set of entities and relationships. The greater expressiveness of process models, as compared to keywords or tables, offers the potential for substantively increased retrieval precision, at the cost of requiring that services be modeled in this more formal way. As we will see below, our preliminary evaluations suggest that the process-based approach offers qualitatively increased retrieval precision, and we will argue that this can be achieved with a reasonable expenditure of service modeling effort.

But first let us understand, in more detail, how process-based retrieval works. Our approach can be viewed as having the following functional architecture (Figure 4):

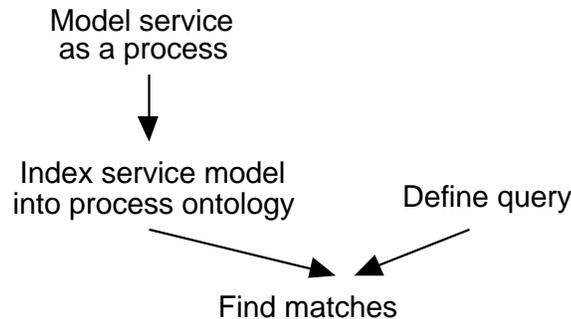


Fig. 4. Functional architecture of process-based service retrieval

We will consider each component of this architecture in the sections below.

3.1 Modeling Services as Process Models

The first step in our approach is to capture service behavior as process models. Why process models? To understand this choice, we need to understand more precisely the causes of imperfect precision (i.e. of false positives). One cause is that a component of the service model is taken to have an unintended *role*. For example, a keyword-based query to find mortgage services that deal with “payment defaults” (a kind of exception) would also match descriptions like “the payment defaults to \$100/month” (an attribute value). The other cause for false positives occurs when a service model is taken to include an unintended *relationship* between components. For example, we may be looking for a mortgage service where insurance is provided for payment defaults, but a keyword search would not distinguish this from a service that provides insurance for the home itself.

The trick to increasing retrieval precision, therefore, comes down to ensuring that the roles and relationships that are meaningful to the user are made explicit in both the query and the service model, so unintended meanings (and therefore false positives) can be avoided. We believe that process-modeling languages are well suited for this. Process modeling languages have been designed to capture the essence of different behaviors in a compact intuitive way, and have become ubiquitous for a very wide range of uses. A de facto consensus has emerged on how to model processes: they all for example use essentially the same primitives (i.e. tasks and subtasks, resources, inputs, outputs, and exceptions).

We use for our purposes the following process modeling formalism (Figure 5):

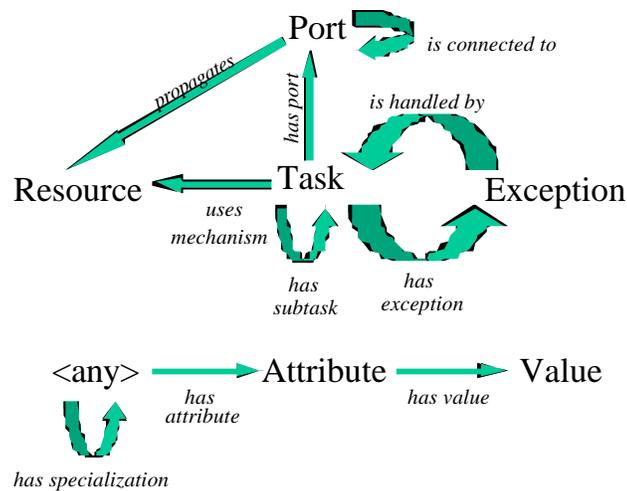


Fig. 5. Process model formalism

The key components of this formalism include:

- **Attributes:** Processes can be annotated with attributes that capture such information as a textual description, typical performance values (e.g. how long a process takes to execute), and so on.
- **Decomposition:** A process can be modeled as a collection of processes that can in turn be broken down (“decomposed”) into sub-processes.
- **Resource Flows:** All process steps can have input and output *ports* through which *resources* flow. One innovation we use is to recognize that processes can be divided into ‘core’ activities as well as those involved in coordinating the flow of resources between core activities [23]. This insight allows us to abstract away details about how sub-processes coordinate with each other, allowing more compact service descriptions without sacrificing significant content.
- **Mechanisms:** Processes can be annotated with the resources they *use* (as opposed to consume or produce). For example, the Internet can serve as a mechanism for a process.
- **Exceptions:** Processes typically have characteristic ways they can fail and, in at least some cases, associated schemes for anticipating and avoiding or detecting and resolving them. This is captured in our approach by annotating processes with their characteristic ‘exceptions’, and mapping these exceptions to processes describing how these exceptions can be handled [24].

Let us consider a simple example to help make this more concrete (Figure 6):

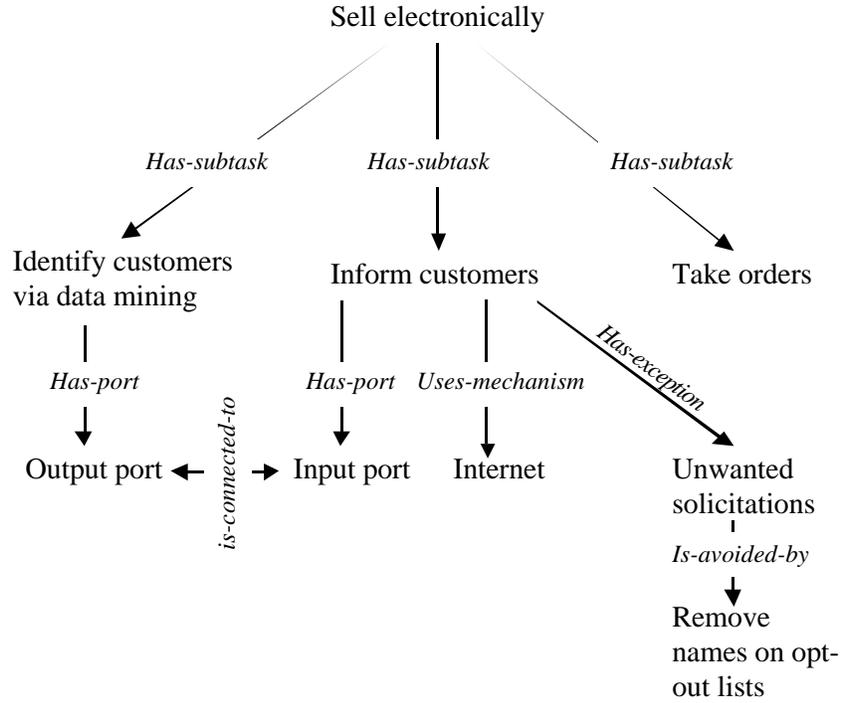


Fig. 6. An example of a process-based service model

This represents the process model for a service for selling items electronically. The plain text items represent entities (such as exceptions, ports and tasks), while the italicized items represent relationships between these entities. The substeps in this service model include ‘identify potential customers via data mining’, ‘inform customers’ (which uses the Internet as a mechanism), and ‘take orders’. The potential exception of sending out unwanted solicitations is avoided by filtering out the names of individuals who have placed their names on ‘opt-out’ lists. Each of the entities can have attributes (not shown) that include their name, description, and so on.

Formally, any database of process descriptions (using the formalism above) can be defined as a typed graph:

$$Ont(Entities, Relationships) \quad (1)$$

where *entities* are the nodes in the graph and *relationships* are the graph edges. Furthermore, the following specifications apply:

- A node can only have one type (\oplus is used to denote the logical exclusive-or operator):

$$x \in Entities \equiv (x \in Task) \oplus (x \in Resource) \oplus (x \in Port) \oplus (x \in Exception) \oplus (x \in Attribute) \oplus (x \in Value) \quad (2)$$

- A relationship can only have one type and it connects nodes of certain types:

$$\begin{aligned}
 r(x, y) \in Relationships \equiv & [(r = Has_specialiation) \wedge (x \in Entities) \wedge (y \in Entities) \wedge ((x, y) \in Has_specialiation)] \oplus \\
 & [(r = Has_subtask) \wedge (x \in Task) \wedge (y \in Task) \wedge ((x, y) \in Has_subtask)] \oplus \\
 & [(r = Has_port) \wedge (x \in Task) \wedge (y \in Port) \wedge ((x, y) \in Has_port)] \oplus \\
 & [(r = Uses_mechanism) \wedge (x \in Task) \wedge (y \in Resource) \wedge ((x, y) \in Uses_mechanism)] \oplus \\
 & [(r = Propagates) \wedge (x \in Port) \wedge (y \in Resource) \wedge ((x, y) \in Propagates)] \oplus \\
 & [(r = Is_handled_by) \wedge (x \in Exception) \wedge (y \in Task) \wedge ((x, y) \in Is_handled_by)] \oplus \\
 & [(r = Has_exception) \wedge (x \in Task) \wedge (y \in Exception) \wedge ((x, y) \in Has_exception)] \oplus \\
 & [(r = Has_subtask) \wedge (x \in Task) \wedge (y \in Task) \wedge ((x, y) \in Has_subtask)] \oplus \\
 & [(r = Has_attribute) \wedge (x \in Entities) \wedge (y \in Attribute) \wedge ((x, y) \in Has_attribute)] \oplus \\
 & [(r = Has_value) \wedge (x \in Attribute) \wedge (y \in Value) \wedge ((x, y) \in Has_value)]
 \end{aligned} \quad (3)$$

This representation is similar, and equivalent in expressiveness, to other full-fledged process modeling languages (e.g. IDEF [25], PIF [26], PSL [27] and CIMOSA [28]) and substantially more expressive than the keyword and table-based languages used in previous service retrieval efforts, by virtue of adding the important concepts of resource flows, task decompositions, and exceptions. It does not however, include primitives oriented at expressing control semantics, i.e. that describe *when* each subtask gets enacted. Such primitives were excluded for two reasons. One is that the bulk of the variation between different process modeling languages occurs in the realm of representing control semantics, and we wanted to begin with a formalism to which a wide range of existing process models could easily be translated. The other reason is that it is our experience to date that most service queries are concerned with *what* a process does, rather than *when* the parts of the process gets enacted.

Modeling service behaviors as process models of course involves manual effort, but we argue this need not be a major barrier. Because process formalisms are so widely used, many of the services we want to model already have process models defined for them. Software applications and business models, for example, are described as flow charts as standard practice. The expertise needed to create such models is widely available. Process ontologies (see below) can reduce the modeling effort involved. Finally, service providers will likely be motivated to create such process models, since they often differentiate themselves in the marketplace by how they provide their services, and process models make this explicit. Process models, finally, enable important uses other than search, such as automatic service composition.

3.2 Indexing Service Models into the Process Ontology

The second step of our approach is to index service models into a process ontology in order to facilitate later retrieval. An ontology consists, in general, of a hierarchy of entity descriptions ranging from the abstract at one end to the specific at the other. Items with similar semantics (e.g. processes with similar functions) appear close to each other, the way books with similar subjects appear close to each other in a library. Indexing a service comes down to placing the associated process model, as well as all of its components (attributes, ports, dependencies, subtasks and exceptions) on the appropriate branch in the ontology. Using an ontology is valuable for several reasons. It can reduce the burden of modelling a service, since one need only find the most similar process model in the ontology and then modify it to model a new service. Ontologies can, in addition, increase recall. Since similar services are co-located, one is apt to find relevant services simply by browsing the ontology near the matches one has already found. In addition, an ontology helps us find matches that are described using different, but semantically equivalent, terminology.

We build for this purpose on the MIT Process Handbook project. The Handbook is a process ontology which has been under development at the Center for Coordination Science (CCS) for the past ten years [23] [29]. The growing Handbook database currently includes roughly 5000 process descriptions ranging over such areas as supply chain logistics, hiring, conflict management, multi-criteria decision making, and so on (Figure 7):

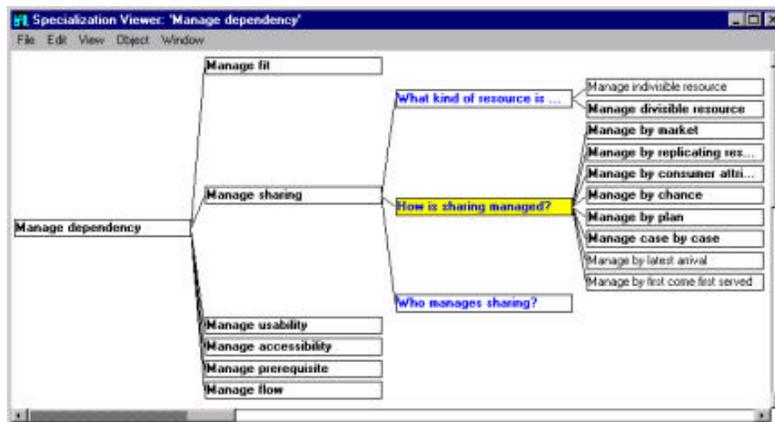


Fig. 7. A fragment of the Handbook process ontology

The Handbook project has developed sophisticated tools for ontology editing that have been refined over years of daily use. Using these tools, most process models can be indexed by a knowledgeable user in a matter of minutes. We believe that the Handbook ontology represents an excellent starting point for indexing many services because it is focused on business processes, which is what a high proportion of such services are likely to address.

3.3 Defining Queries

It is of course imaginable that we could do without queries entirely once services have been indexed into an ontology. One could simply browse the ontology to find the services that one is interested in, as in [30]. Our experience suggests however that browsing can be slow and difficult for all except the most experienced users. This problem is likely to be exacerbated when, as with online services, the space of services is large and dynamic.

To address this challenge we have defined a query language called PQL (the Process Query Language) designed for retrieving process models indexed in an ontology. Process models can be straightforwardly viewed as entity-relationship diagrams made up of entities like tasks characterized by attributes and connected by relationships like ‘has-subtask’. PQL queries are built up as combinations of three primitive clause types that check for these elements:

- Entity <entity> isa <entity type>
- Relation <source entity> <relationship type> <target entity> [*]
- Attribute <attribute> of <entity> {includes | equals} <value>

The ‘entity’ clause matches any entity of a given type (the entity types include task, resource, port and so on). The ‘relation’ clause matches any relationship of a given type between two entities (the relationship types include has-subtask, has-specialization, has-port, and so on). The optional asterisk finds the transitive closure of this relationship. The ‘attribute’ clause looks for entities with attributes that have given values. Any bracketed item <> can be replaced by a variable (with the format ?<string>) that is bound to the matching entity and passed to subsequent query clauses.

We have also found it useful to include an operator for grouping clauses into sub-queries:

- When {exists | does-not-exist} <query>

The ‘when’ clause enables further pruning by submitting the matches returned by a sub-query to a predicate. At present, only two built-in predicates have been defined but this will likely expand

Let us consider a simple example to help make this more concrete. The query below searches for a sales service that uses the internet to inform customers:

```
attribute "Name" of ?sell includes "sell"
relation ?sell has-specialization ?process *
when exists (relation ?process has-subtask ?subtask *
             attribute "Name" of ?subtask includes "inform"
             attribute "Description" of ?subtask includes "internet")
```

The first clause searches for a processes in the ontology whose name includes “sell”, the second finds all specializations of this, and the third checks if any subtasks of these services are “inform” processes with “internet” in their description. A PQL query is thus equivalent to a sub-graph pattern, and any search for a process model can then be treated as finding the nodes of type *task* which match the graph pattern that represents the query.

The three clause types of PQL, and their variants, can be formalized as follows:

Relation x rel-type y

is defined as:

$(x,y) \in \text{rel-type}$

where $\text{rel-type} \in \{\text{has-specialization, has-subtask, has-exception} \dots\}$

Relation x rel-type y *

is formalized using a fixpoint/recursive definition, as:

$((x, y) \in \text{rel_type}) \vee (\exists z : ((x,z) \in \text{rel_type}) \wedge \text{rel_type}(z,y)^*)$

Entity entity isa entity-type

is defined as:

$\text{entity} \in \text{entity-type}$

where $\text{entity-type} \in \{\text{Task, Port, Resource, Exception, Attribute, Value}\}$

Attribute attribute of entity equals value

is shorthand for two relationships, as follows:

$\text{has_attribute}(\text{entity}, \text{attribute}) \wedge \text{has_value}(\text{attribute}, \text{value})$

Attribute attribute of entity includes value

is defined as:

$\text{has_attribute}(\text{entity}, \text{attribute}) \wedge \text{has_value}(\text{attribute}, v1) \wedge \text{IsSubString}(\text{value}, v1)$

Note that PQL includes built-in functions, such as `IsSubString`, comparable to those in other query languages such as SQL. If any of the parameters to a predicate are preceded by a question mark (e.g., `?y`), then it denotes a variable that needs to be bound to a value from the database/model that can fulfill its place.

The ‘when’ construct serves two roles. If used with the “exists” operator then it simply groups sub-queries in an intuitive way, and does not add any expressive power to PQL. For example:

Relation ?x Has_subtask ?y

When exists ((Relation ?y Has_subtask a))

is formalized equivalently with or without the “when” operator, as:

$\text{Has_subtask}(?x, ?y) \wedge \text{Has_subtask}(?y, a)$

If the when-statement is used with the “does-not-exist” option then it will only return a result if `<query>` does not. This introduces a form of *negation* into PQL, so:

When does-not-exist query

is defined as

$\neg \exists x_{i=1..k}: x_{i=1..k} \in \text{Entities} : \langle \text{query} \rangle$

where: $x_{i=1..k}$ are the unbound variables in query.

The question of how to add negation to a query language is a non-trivial issue, as it may have major implications on its computational tractability if the language includes recursive (or iterative) features or fixpoint semantics. PQL includes such semantics with its `*`-modifier. As will become obvious in section 5 below, the type of negation introduced here is consistent with an inflationary fixpoint approach, ensuring that the resulting language is bounded by polynomial time.

As a final example, let us consider how our original example PQL query is formalized:

$\text{Has_attribute}(?sell, \text{“Name”}) \wedge \text{Has_value}(\text{“Name”}, ?v1) \wedge \text{IsSubString}(\text{“sell”}, ?v1) \wedge$
 $\text{Has_Specialization}(?sell, ?process) \wedge$
 $\text{Has_attribute}(?process, \text{“service?”}) \wedge \text{Has_value}(\text{“service?”}, \text{“yes”}) \wedge$
 $(\text{Has_subtask}(?process, ?subtask) \wedge$
 $\text{Has_attribute}(?subtask, \text{“Name”}) \wedge \text{Has_value}(\text{“Name”}, ?v2) \wedge \text{IsSubString}(\text{“inform”}, ?v2) \wedge$
 $\text{Has_attribute}(?subtask, \text{“Description”}) \wedge \text{Has_value}(\text{“Description”}, ?v3) \wedge \text{IsSubString}(\text{“inform”}, ?v3))$

PQL has been used successfully to represent a wide range of queries drawn from many different domains. Some other examples include “find a loan process that uses the internet, takes real estate as collateral, and has loan default insurance”, “find a way to manufacture bicycles in a way that is cheaper than average and does not use expensive machinery”, “find all processes that take oil as an input and are prone to cause environmental damage”, and so on. Our preliminary assessment is that PQL is sufficiently expressive to capture all queries describable in process-oriented terms.

3.4 Finding Matches

The algorithm for retrieving matches given a PQL query is straightforward. The clauses in the PQL query are tried in order, each clause executed in the variable binding environment accumulated from the previous clauses. The bindings that survive to the end represent the matching services. Query performance can be increased by using such well-known techniques as query clause re-ordering. While we have not yet evaluated PQL’s performance in detail yet, we do show (see below) that queries are within polynomial time complexity.

4 Empirical Evaluation

An initial version of the PQL interpreter has been implemented, and we have performed some preliminary evaluations of its precision and recall compared to existing (keyword and table-based) approaches. The following scheme was used for all of the evaluations described below. The roughly 5000 processes in the Process Handbook process ontology were treated as service models, which is reasonable since they all represent functions used in business contexts and many could imaginably be performed remotely. We then defined keyword and process-based queries that use the same keywords, operate over the same database of service models, and differ only in whether they use the role and relationship information encoded in the service models. We did not define a separate set of table-based service models and queries for this evaluation because, from the standpoint of retrieval precision, the keyword and table-based approaches are equivalent. The fact that table-based models differentiate name and description attributes does not help since descriptions almost invariably reprise the keywords included in the service name, and none of the queries we used made use of I/O specifications. In any case, if we had used queries that refer to such I/O specs, it would not change the relative precision of table- and process-based queries since both can use I/O information. We tested simple keyword search as well as TFIDF, the latter because its potentially greater effective precision makes it a dominant scheme for keyword-based search. All the queries in our evaluation had perfect recall, because of the consistency in the use of keywords in the process descriptions. While we do not anticipate that process-based search will differ significantly in recall from keyword and table-based, this remains a subject for future evaluation. The queries below, clearly, are only illustrative, since a complete evaluation would require executing a representative range of many queries.

Since our goal was to determine whether process-based retrieval improves on existing approaches, our evaluation focused on the value of the additional information captured by process-based service models as compared to keyword- and table-based models. This additional information falls into five categories: task decompositions, port connectivity, exception handling, task mechanisms, and specializations. We examine each category in the sections below.

4.1 Task Decomposition

Our process-based service model allows us to explicitly describe the subtasks that make up a service's behavior. This can help avoid confounding information that refers to different subtasks. Imagine, for example, that we are searching for a sales service that informs customers using the internet. We can frame this query as follows:

Table 1. Query types and actual Queries

| <i>Type</i> | <i>Query</i> |
|---------------|---|
| Keyword-based | "Sell" "inform" "internet" |
| Process-based | attribute "Name" of ?service includes "sell" when exists (relation ?service has-subtask ?subtask * attribute "Name" of ?subtask includes "inform" attribute ?attr of ?subtask includes "internet") |

The keyword and table-based service models are not able to distinguish cases where "inform" and "internet" (or their synonyms) belong to the same subtask from cases where these keywords belong to different subtasks (and thus are probably not relevant). We would thus predict false positives and therefore lower precision for these approaches, and this is in fact what happens. There were 13 correct matches for this query, including such processes as "Sell travel services via electronic auction", "Sell books via electronic store" and so on. The PQL query had 13 correct matches out of the 18 it returned, for a precision of 72%. A simple keyword-based search had 280 returns, for a precision of roughly 5%. TFIDF did not improve much upon simple keyword search in this case: its precision reached a maximum of 6% (at match 163), and its overall precision was lower because it allowed partial matches and therefore generated more total returns,

4.2 Task Mechanisms

Our process-based service model allows us to describe the mechanisms used by a task, thereby avoiding false positives due to the appearance of the same keyword with a different role. We can, for example, refine the PQL query given above so that it only matches services where the keyword "internet" appears as a mechanism (the added clauses are bold type):

```

attribute "Name" of ?service includes "sell"
when exists (relation ?service has-subtask ?subtask *
  attribute "Name" of ?subtask includes "inform"
  relation ?subtask uses-mechanism ?mechanism
  attribute "Name" of ?mechanism includes "Internet")

```

This query had 13 correct matches, as above, but this time out of 16 responses, for an improved precision of 81%.

4.3 Specialization

The has-specialization relationship enabled by our inclusion of a process ontology can be used to avoid false positives by ensuring that the service, and its components, have the semantics that we desire. For example, we can use this to refine the query presented above to only accept services whose subtask is a specialization of the generic "Inform" task, thereby pruning out services with subtasks that include the string "inform" in their name for unrelated reasons (e.g. the subtask named "Collect configuration information using Internet"):

```

attribute "Name" of ?service includes "sell"
when exists (relation ?service has-subtask ?subtask *
  attribute "Name" of ?subtask includes "inform"
  relation ?subtask uses-mechanism ?mechanism
  attribute "Name" of ?mechanism includes "Internet"
  attribute "Name" of ?class equals "Inform"
  relation ?class has-specialization ?subtask)

```

With this refinement, the query returns 13 correct matches out of 13 total, for an accuracy of 100%.

Similar ontologies have of course been made available for table-based service retrieval engines. UDDI, for example, provides the UNSPSC taxonomy of product and service categories and the NAICS taxonomy of industry codes, among others. The particular value of the ontology we utilize is that it captures functions that a business might require at a much finer grain than the taxonomies mentioned above. We believe this will be helpful for service retrieval since many queries will, no doubt, be looking for services to support business functions. At the time of writing our database did not categorize services using these other taxonomies, so we were unable to evaluate their relative merits.

4.4 Exception Handling

Our process-based service model allows us to explicitly delineate the exceptions faced by a service, as well as the handlers available for dealing with each exception. Imagine, for example, that we wish to find a sales service that informs customers via the internet but avoids the exception of sending unwanted solicitations (e.g. by filtering out the names that appear on "opt-out" lists). We can, for this purpose, refine the query described above as follows:

Table 2. Query types and Query for exception handling Query

| <i>Type</i> | <i>Query</i> |
|---------------|--|
| Keyword-based | "Sell" "inform" "internet" "avoid" "unwanted" "opt-out" |
| Process-based | <pre> attribute "Name" of ?service includes "sell" when exists (relation ?service has-subtask ?subtask * attribute "Name" of ?subtask includes "inform" relation ?subtask uses-mechanism ?mechanism attribute "Name" of ?mechanism includes "Internet" attribute "Name" of ?class equals "Inform" relation ?class has-specialization ?subtask relation ?subtask has-exception ?exception attribute "Name" of ?exception includes "unwanted" relation ?exception is-avoided-by ?handler relation ?attr of ?handler includes "opt-out") </pre> |

We would expect the keyword- and table-based models to incur false positives by finding services that have the same keywords in different roles, that have that exception but do not have a handler for it, or that use a different handler (e.g. allowing recipients to remove their name from subsequent solicitations) for the same exception. In this case, there was one correct match. PQL returned only that item, for a precision of 100%. Keyword-based search returned 188 matches (0.53% precision), and TFIDF did not do any better, returning 248 documents, with a maximum precision at 0.4%.

4.5 Port Connectivity

The final category of information uniquely provided by process-based service models is port connectivity, which captures the resource flow relationships between tasks. We may, for example, want a service that generates the lists of potential customers to inform by applying data-mining techniques, which implies that the output of a data mining subtask is an input to the inform customers subtask. This would imply a PQL query like the following:

```
attribute "Name" of ?sell includes "sell"
when exists (relation ?process has-subtask ?sub1
  attribute "Name" of ?sub1 includes "inform"
  relation ?sub1 has-port ?port1
  entity ?port1 isa input-port
  relation ?process has-subtask ?sub2
  attribute "Name" of ?sub includes "mining"
  relation ?sub2 has-port ?port2
  entity ?port2 isa output-port
  relation ?port1 is-connected-to ?port2)
```

A query like this can avoid false positives wherein a data-mining subtask exists in the service model, but it does not provide information to the inform customers step. The data-mining subtask may be applied instead, for example, to the database of sales generated by this service. We would therefore expect the keyword- and slot-based retrieval queries to demonstrate lower precision than PQL. At the time of writing we were unable to evaluate this because the Process Handbook ontology did not include sufficient port connectivity information; this will be addressed in future work.

4.6 Conclusions

While a wider range of queries and services needs to be evaluated, these test cases strongly suggest that the greater expressiveness of process-based service models can in fact result in qualitatively higher retrieval precision. Even a PQL query that only took advantage of the subtask relationships in the process-based models produced retrieval precision more than 10 times greater than keyword-based approaches. The relative advantage of PQL, moreover, increased radically as the number of relationships specified in the query increased.

5 Complexity Analysis

One of the major considerations for any retrieval capability is that it must return answers in a timely way. The speed at which queries get returned should be comparable to that of existing document retrieval mechanisms, which manage to search millions of documents in seconds. Even though our experience with the prototype implementation has been favorable (i.e., queries generally take several seconds at most, even though our implementation does not exploit well-known performance-enhancing techniques such as reverse indices or query optimization) there is still the question of how the performance will scale with the size of the database. The computational complexity of PQL can be most easily assessed by mapping it to formal query languages with known computational complexity. We will start by showing that PQL without the *-modifier can be mapped to first-order query languages like CALC whose complexity is in QLOGSPACE. We then extend the analysis by mapping PQL with the *-modifier to a DATALOG-type language whose computational complexity is polynomial. These are good results: polynomial complexity implies that the computation needed to enact a PQL query is bounded by a polynomial function in the size of the service model database. We can, thus, expect performance comparable to most database query languages.

5.2 The complexity of PQL without the *-modifier is in QLOGSPACE

CALC can be loosely defined as all the queries that can be defined using first-order calculus (i.e., any query that can be defined using $\neg, \wedge, \vee, \forall, \exists$, without recursion/fixpoints) and has been shown to be in QLOGSPACE [31].

Theorem #1: Any PQL query (without the *-modifier) can be mapped to CALC

Proof: The Proof basically follows from the formal definition of the PQL clauses. All the relationships of the PQL-model are basically relations. For example, *Has_type* is basically a binary relation where both parameters are from the domain *Entities* (also defined in the data-model). Also, any PQL (without the *-modifier) clause can be written as a conjunction of Relationship assertions, basically limiting the scope of the relationship. Expressed formally:

1. PQL as a conjunction of relational assertions:

A basic PQL query is defined as

$$r_1(p1_1, p2_1) \wedge r_2(p1_2, p2_2) \wedge \dots \wedge r_k(p1_k, p2_k),$$

where 1) all $r_{i, i=1..k} \in \text{Relationships}$

2) $pX_{i, i=1..k}$ are elements of the respective domains

3) some of the pX_i are bound to values

4) there might be some X, Y and i, j where $pX_i = pY_j$

Which can be rewritten as a CALC query

$$\{(pX_i, \dots) \mid \exists pX_i, \dots : r_1(p1_1, p2_1) \wedge r_2(p1_2, p2_2) \wedge \dots \wedge r_k(p1_k, p2_k)\}$$

where the X and i denote the indices to unbound parameters.

2. Grouping does not any expressive power to PQL. Hence, it does not need to be mapped to CALC.

3. Grouping with negation adds the element of the negation of an existential quantifier to PQL, which is contained in the CALC language specification

□

Given Theorem #1 and the fact that CALC is in QLOGSPACE it follows that PQL (without the *-modifier) is in QLOGSPACE. As we are mainly interested in the time properties and QLOGSPACE is contained in polynomial time we can deduce that the time complexity of PQL (without *-modifier) is at worst polynomial.

5.2 The complexity of PQL is in polynomial time

The question remains what to do about the *-modifier. One option is to pre-compute the typed transitive hull for each of the relationships used in the data model, so that every relation would have a respective starred relation that would be its' transitive closure. Given this, any starred query would be in QLOGSPACE. There remains the question of the complexity of updating the starred relations when adding a new element.

Theorem #2: Updating the starred relationship on inserting a new element into the relationship is at worst linear in the size of the starred relationship.

Proof (sketch): When a new element **x** gets added to as being related to element **a** to relation R (i.e., as (a, x)) then all that would need to be done is to, first, add (a,x) to R and, second, query R* for all elements where some element y is related to a as follows (y, a) and then update R* for each of these y's with (y,x). Spelled out as an algorithm:

FOR EACH (y,z) in R*

IF z = a THEN ADD (y, x) to R*

NEXT

Which is at worst linear in the size of R*

□

Alternatively, we could evaluate the starred relationship at run-time, which is equivalent to computing the reachability of one node on a graph from another node, which is computable in DATALOG as follows:

$$\text{Reachable}(x, y) \leftarrow R(x, y)$$

$$\text{Reachable}(x, y) \leftarrow \text{Reachable}(x, u), R(u, y)$$

This is basically a different notation for the definition of the *-modifier above. Consequently, as reachability can be mapped to DATALOG and DATALOG has been shown to be in polynomial time ([31] pp. 437, 343-355), the evaluation of any PQL-query (including the *-modifier) can be computed in polynomial time.

As mentioned above one problem remains. Standard DATALOG does not contain negation. If we add negation the semantic interpretation of DATALOG becomes unclear. One has to decide whether one uses strictly inflationary fixpoint semantics, leading to a query language bounded in polynomial *time*, or whether one uses non-inflationary fixpoint semantics, leading to a language bounded by polynomial *space*. For computational tractability the former would be preferred. In our particular case we are lucky. First, the only part of PQL that uses fixpoint semantics is the computation of the starred relationships. These can be seen as pure DATALOG queries (i.e, without negation) embedded in a CALC query. Second, our language adheres to the inflationary semantics as it does not remove previously added result bindings when calculating the fixpoint. Hence it is contained in $DATALOG_{\neg}$, which is bounded by polynomial time.

6 Contributions of this Work

High retrieval service precision is widely recognised as a critical enabler for important uses that range from finding useful software components or applications, to finding best practice models, to tracking down people or organisations with the skills you need. Our work can be viewed as representing a new class of service retrieval technique that helps achieve these goals (Figure 8):

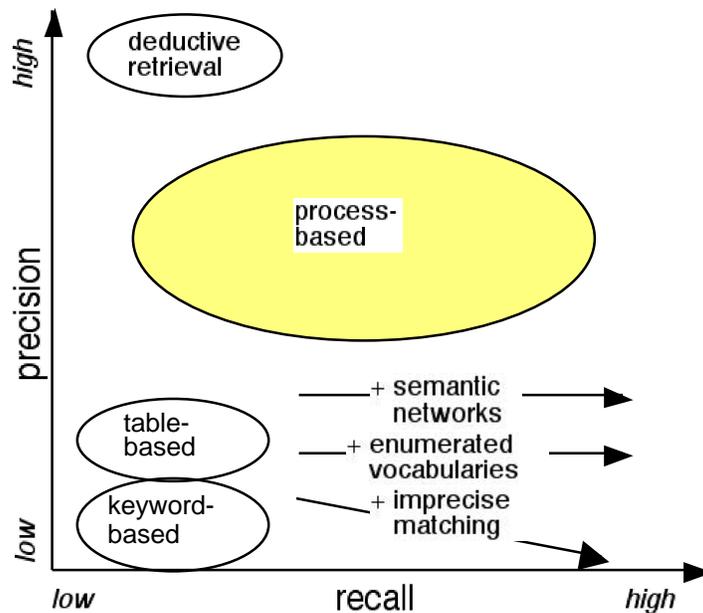


Fig. 8. The contribution of process-based service retrieval technology

Our evaluations to date suggest that process-based queries produce retrieval precision qualitatively greater than that of existing service retrieval approaches, while retaining polynomial complexity for query enactment. This work represents a significant contribution to work on pattern matching over generic graph-like representations, such as graph grammars [32], object-oriented query languages [33], and XQuery (a query language for XML; see <http://www.w3.org/TR/xquery/>). The unique value of our work comes from exploiting the more constrained semantics of process models to enable higher-precision service retrieval.

7 Next Steps

While our preliminary results are promising, many important challenges remain:

- *Evaluation:* PQL needs to be evaluated for a wide range of queries in order to assess its precision and recall performance as compared to existing approaches, and to suggest refinements in the language and associated

interpreter. An important issue will be examining the tradeoff between the expressiveness of the service models, the accuracy/completeness of service model indexing, and the resulting precision/recall performance.

- *Increased recall*: PQL currently makes no use of standard recall-enhancing techniques such as synonym databases or inexact/partial matches. One immediate step will involve enhancing the PQL interpreter with a semantic net such as WordNet [34] that represents synonym and other useful relationships. This promises to be a powerful combination because a process-oriented query gives us crucial additional information about the role of the keywords in the query (e.g. a keyword in a task name is probably a verb, and a keyword in a mechanism name is probably a noun). Another key issue involves *modeling differences*. It is likely that in at least some cases a service may be modeled in a way that is semantically equivalent to but nevertheless does not syntactically match a given PQL query. We plan to explore for this purpose the use of query mutation operators that can modify a given PQL query in a way that (largely) preserves its semantics (e.g. by generalizing a type specification) while allowing a wider range of service matches.
- *Automated indexing*. A key criterion for a successful service retrieval approach is minimizing the manual effort involved in listing new services with the search engine. Ideally services can be classified automatically. Previous efforts for automatic service classification have used similarity metrics based mainly on word frequency statistics [35] [36] [37]. We plan to augment such approaches using the taxonomic reasoning and graph-theoretic similarity measures made possible by our approach to service modeling.
- *User interface*. PQL, like most query languages, is fairly verbose and requires that users be familiar with its syntax. An important part of our work will be to develop more intuitive interfaces suitable for human users, for example based on flowcharts or natural language.

References

1. Bakos, J.Y., *Reducing Buyer Search Costs: Implications for Electronic Marketplaces*. Management Science, 1997. 43.
2. Mili, H., F. Mili, and A. Mili, *Reusing software: issues and research directions*. IEEE Transactions on Software Engineering, 1995. 21(6): p. 528-62.
3. Salton, G., G. Salton, and M.J. McGill, *Introduction to Modern Information Retrieval*. 1983, New York: McGraw-Hill.
4. Salton, G. and M.J. McGill, *Introduction to modern information retrieval*. McGraw-Hill computer science series. 1983, New York: McGraw-Hill. xv, 448.
5. Prieto-Diaz, R., *Implementing faceted classification for software reuse*. 12th International Conference on Software Engineering, 1990. 9: p. 300-4.
6. Magnini, B., *Use of a lexical knowledge base for information access systems*. International Journal of Theoretical & Applied Issues in Specialized Communication, 1999. 5(2): p. 203-228.
7. Henninger, S., *Information access tools for software reuse*. Journal of Systems & Software, 1995. 30(3): p. 231-47.
8. Fernandez-Chamizo, C., et al., *Case-based retrieval of software components*. Expert Systems with Applications, 1995. 9(3): p. 397-421.
9. Fugini, M.G. and S. Faustle. *Retrieval of reusable components in a development information system*. in *Second International Workshop on Software Reusability*. 1993: IEEE Press.
10. Devanbu, P., et al., *LaSSIE: a knowledge-based software information system*. Communications of the ACM, 1991. 34(5): p. 34-49.
11. ESPEAK, *Hewlett Packard's Service Framework Specification*. 2000, HP Inc.
12. Raman, R., M. Livny, and M. Solomon, *Matchmaking: an extensible framework for distributed resource management*. Cluster Computing, 1999. 2(2): p. 129-38.
13. Richard, G.G., *Service advertisement and discovery: enabling universal device cooperation*. IEEE Internet Computing, 2000. 4(5): p. 18-26.
14. Sycara, K., et al. *Matchmaking Among Heterogeneous Agents on the Internet*. in *AAAI Symposium on Intelligent Agents in Cyberspace*. 1999: AAAI Press.
15. Fensel, D. *An Ontology-based Broker: Making Problem-Solving Method Reuse Work*. in *Workshop on Problem-Solving Methods for Knowledge-based Systems at the 15th International Joint Conference on AI (IJCAI-97)*. 1997. Nagoya, Japan.
16. Mengendorfer, S. and P. Manhart. *A Knowledge And Deduction Based Software Retrieval Tool*. in *6th Annual Knowledge-Based Software Engineering Conference*. 1991: IEEE Press.
17. Chen, P., R. Hennicker, and M. Jarke. *On the retrieval of reusable software components*. in *Proceedings Advances in Software Reuse. Selected Papers from the Second International Workshop on Software Reusability*. 1993.

18. Kuokka, D.R. and L.T. Harada, *Issues and extensions for information matchmaking protocols*. International Journal of Cooperative Information Systems, 1996. **5**: p. 2-3.
19. Podgurski, A. and L. Pierce, *Retrieving reusable software by sampling behavior*. ACM Transactions on Software Engineering & Methodology, 1993. **2**(3): p. 286-303.
20. Hall, R.j. *Generalized behavior-based retrieval (from a software reuse library)*. in *15th International Conference on Software Engineering*. 1993.
21. Park, Y., *Software retrieval by samples using concept analysis*. Journal of Systems & Software, 2000. **54**(3): p. 179-83.
22. Klein, M. and A. Bernstein. *Searching for Services on the Semantic Web using Process Ontologies*. in *The First Semantic Web Working Symposium (SWWS-1)*. 2001. Stanford, CA USA.
23. Malone, T.W. and K. Crowston, *The interdisciplinary study of coordination*. ACM Computing Surveys, 1994. **26**(1): p. 87-119.
24. Klein, M. and C. Dellarocas, *A Knowledge-Based Approach to Handling Exceptions in Workflow Systems*. Journal of Computer-Supported Collaborative Work. Special Issue on Adaptive Workflow Systems., 2000. **9**(3/4).
25. NIST, *Integrated Definition for Function Modeling (IDEF0)*. 1993, National Institute of Standards and Technology.
26. Lee, J. and M. Gruninger, *The Process Interchange Format "The Process Interchange Format and Framework v.1.2*. Knowledge Engineering Review, 1998(March).
27. Schlenoff, C., et al., *The essence of the process specification language*. Transactions of the Society for Computer Simulation, 1999. **16**(4): p. 204-16.
28. Kosanke, K., *CIMOSA: Open System Architecture for CIM*. 1993: Springer Verlag.
29. Malone, T.W., et al., *Tools for inventing organizations: Toward a handbook of organizational processes*. Management Science, 1999. **45**(3): p. 425-443.
30. Latour, L. and E. Johnson. *Seer: a graphical retrieval system for reusable Ada software modules*. in *Third International IEEE Conference on Ada Applications and Environments*. 1988: IEEE Comput. Soc. Press.
31. Abiteboul, S., R. Hull, and V. Vianu, *Foundations of Databases*. 1995, Reading, MA: Addison-Wesley Publishing.
32. Rozenberg, G., ed. *Handbook of Graph Grammars and Computing by Graph Transformation*. Vol. 3. 1999, World Scientific.
33. Kim, W., *Introduction to Object-Oriented Databases*. 1990, Cambridge, MA: MIT Press.
34. Fellbaum, C., ed. *WordNet: An Electronic Lexical Database*. 1998, MIT Press: Cambridge MA USA.
35. Frakes, W.b. and B.a. Nejme, *Software reuse through information retrieval*. Proceedings of the Twentieth Hawaii International Conference on System Sciences, 1987. **2**: p. 6-9.
36. Maarek, Y.s., D.M. Berry, and G.e. Kaiser, *An information retrieval approach for automatically constructing software libraries*. IEEE Transactions on Software Engineering, 1991. **17**(8): p. 800-13.
37. Girardi, M.R. and B. Ibrahim, *Using English to retrieve software*. Journal of Systems & Software, 1995. **30**(3): p. 249-70.