Computing and Informatics, Vol. 32, 2013, 827–843

PAY-AS-YOU-GO SOFTWARE ARTIFACTS MANAGEMENT

Ying Pan

College of Computer and Information Engineering
Guangxi Teachers Education University
Nanning, 530023 P.R. China
e-mail: panying6@mail3.sysu.edu.cn

Yong Tang

School of Computer Science
South China Normal University
Guangzhou, 510631 P.R. China
e-mail: ytang@scnu.edu.cn

Communicated by Ján Paralič

Abstract. One of the major challenges in software engineering research is to manage software artifacts effectively. However, software artifacts are often changed during software development, the full, one-time integration technique is not feasible to manage such heterogeneity and evolving data. In this paper, we concern about the application of dataspace techniques, which emphasize the idea of pay-as-you-go data management, to software artifacts management. To this end, we present a loosely structured data model based on the current dataspace models to describe software artifacts, and a strategy to query this model. We also present how to gradually add semantics to query processing for improving the precision and recall of query results. Furthermore, the validity of our work is proved by experiment. Finally, the differences between our work and traditional work are discussed.

Keywords: Software artifacts management, dataspace, pay-as-you-go, software engineering

Mathematics Subject Classification 2010: 68P10

1 INTRODUCTION

One of the major challenges in software engineering research is to manage software artifacts effectively. Extensive efforts have been brought forth to address the issue. For example, different data models (such as XML-based source code representation [1, 2] and graph model [3]) have been proposed to represent software artifacts for managing data effectively. In addition, Semantic Web technologies (e.g., ontology) are proposed to enhance semantic software artifacts descriptions [4, 5]. Utilizing the available ontologies and ontology reasoners, the approaches and tools can provide semantic-based retrieval on software artifacts at the semantic level.

Despite successes, to the best of our knowledge, the works mentioned above pay little attention to managing software artifacts in a gradually, pay-as-you-go (PAYG) fashion. These works involve traditional database technologies or data integration technologies which require hard up-front investment to define a schema for the data before the useful services are provided. That is, they manage artifacts in a pay-before-you-go (PBYG) fashion. The up-front investment is high-cost in a PBYG fashion, because it is a difficult work to describe all software artifacts into the strict data models which enforce a schema over the data. Furthermore, before semantic retrieval is supported, semantic-based approaches require full semantic integration of the data sources; but, it is difficult to understand the data and fully create these semantic mappings immediately. Moreover, software artifacts evolve over time, and then artifact descriptions need to be changed accordingly. Therefore, the full, one-time integration technique is not feasible to manage such heterogeneity and evolving data, which should be integrated and managed in a PAYG fashion.

A new style of PAYG data management called dataspace [6] addresses the above challenges. Dataspace offers best-effort services on heterogeneity data without requiring the up-front effort, and automatically enhance services over time. In this paper we concern about the application of dataspace techniques to software artifacts management. The work presented in this paper is based on our previous work [7], which presents a framework of a dataspace system for software artifacts management. The new contributions of this paper can be summarized as follows:

1. We present a model Software Artifacts Graph (SAG) based on the current dataspace data models. SAG is a very loosely structured data model which represents software artifacts and the relationships among them. Moreover, SAG may be constructed in a PAYG, ongoing fashion.
2. We present an approach to query SAG, and introduce the basic steps of the approach, including query translation and model transformations between SAG and the dataspace model.
3. We introduce the technique to gradually add semantics to query processing. Furthermore, we show how to combine this technique with keyword- and structure-based query for improving the precision and recall of query results.

The rest of this paper is organized as follows. The next section introduces the dataspace techniques which are relevant to our research. Section 3 introduces SAG and its properties. Section 4 shows how to query SAG. Section 5 discusses semantic addition during query processing. Section 6 shows the experimental results and Section 7 evaluates our work in comparison to related work. Finally, Section 8 concludes the paper and outlines our future work.

2 DATASPACE TECHNIQUES

Dataspace was put forward in 2005 SIGMOD [6]. A dataspace contains all information relevant to a particular organization regardless of its location and format, and models any kind of relationships between individual data sources [8].

Dataspace Management System (DSMS) provides the required services (e.g., search and query) over dataspaces without requiring expensive semantic integration. That is, DSMS provides an abstraction for accessing, understanding, managing, and querying over all autonomous and heterogeneous data sources, and organizing their data over time in an incremental, PAYG approach [6].

One of the fundamental shortcomings of existing information integration systems and database systems is the long setup time required. In sharp contrast to them, DSMS emphasizes the idea of PAYG data management: it offers best-effort services on data without requiring the upfront effort, and improves the services as more investment is made into identifying semantic relationships [9]. That is, dataspace system offers varying levels of services according to the situation. For example, dataspace system should begin by offering base services such as keyword search over a collection of data even before semantic mappings are created, and return best-effort or approximate answers to users. Over time, if a higher level of semantic integration is required, users (or database administrators) decide where and when it is worthwhile to invest more efforts in providing semantic mappings or other kind of semantic constructs that improve the accuracy of query results.

Personal information management has received considerable attention in dataspaces communities, with some famous Personal Dataspace Management Systems (e.g., iMeMex [10]) being developed. iDM [11], the iMeMex data model, represents structured, semi-structured and unstructured data into a resource view graph, which is a logical representation of personal information. One important aspect of iDM is that it is lazily computed, i.e., all nodes and connections in the graph may be computed dynamically as deemed necessary. On top of iDM, the query language iQL is proposed to allow users to write intuitive keyword searches with structural restrictions. Moreover, iTrails [12] technique which provides a mapping from one query to another is proposed in iMeMex. With iTrails technique, users may gradually provide lightweight semantics to query processing, and then the accuracy of query results is improved.

3 REPRESENTING SOFTWARE ARTIFACTS

3.1 iDM and AVG Data Model

iDM [11] represents all personal information (e.g., MS Word or other Office documents, XML, relational data, file content, folder hierarchies, email, data streams and RSS feeds) into a logical graph. In recent years, some dataspace data models based on iDM are proposed for different aims [12, 13]; we denote the model mentioned in [13] as attribute-value graph model, which is summarized in the following definition.

Definition 1 (Attribute-Value Graph, AVG)**.** The data on the dataspace is represented by an attribute-value graph $G := (N, E)$, where N is a set of nodes, each node N_i is a set of attribute-value pairs. E is a set of edges (N_i, N_j, L), where L is a label and $N_i, N_j \in N, i \neq j$.

Each resource view in iDM can be seen as a set of attribute-value pairs [14], that is, iDM and AVG are equivalent. Therefore, the techniques in iMeMex system support both iDM and AVG.

3.2 Software Artifacts Graph

All the current dataspace data models, including iDM and AVG, have not involved the issue of the representation of software artifacts, especially source code. Although iDM and AVG may describe software artifacts, they are not much suitable for software artifacts management. iDM only describes the sequence relationship of the data sources; although it may describe complex relationships by describing the sequence relationship, users still have difficulty to model complex relationships. As to AVG, it can not represent the relative order among the connections established between nodes.

For better describing the software artifacts, we propose a data model SAG based on the model iDM and AVG, with extending the concept of edge component to describe more complex data sources and their relationships. SAG is summarized in the following definition.

Definition 2 (Software Artifacts Graph, SAG)**.** Software artifacts are represented by a logical software artifacts graph $G := (V, E)$, where V is a set of nodes, each node V_i is a 3-tuple (η_i, τ_i, χ_i). η_i is the name of V_i. τ_i is a 2-tuple (W, T), where W represents the attributes of V_i and T represents their corresponding values. χ_i records the content of V_i; it may represent arbitrary unstructured content. E is a set of edges between V_i and other nodes. E is a 2-tuple (R, L), where R represents nodes which are related to V_i, and L, the label of the edge, represents the relationship between V_i and R. R is also a 2-tuple (A, B), where A is a set of nodes and B is an ordered sequence of nodes, and $A \cap B = \emptyset$[1].

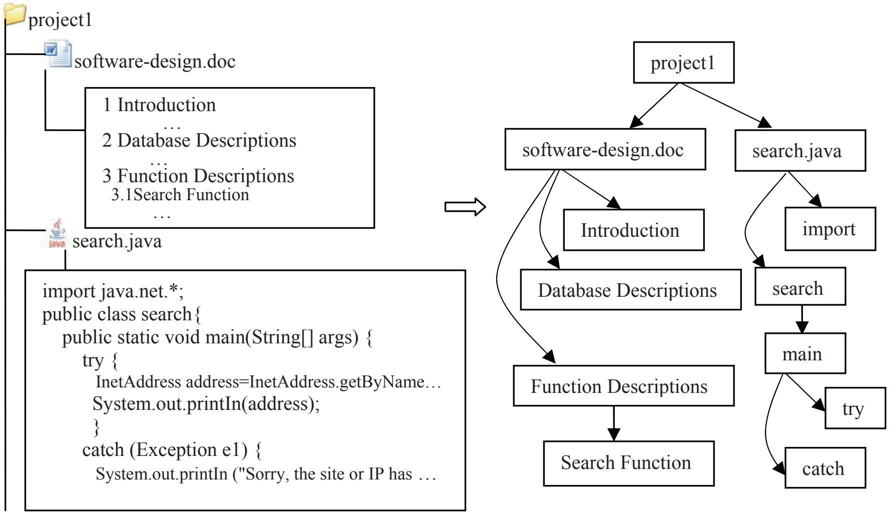
[1] We use the symbols $\cap$ just like the literature [11] to denote intersection not only between sets, but also between a set and a sequence.

If there exists an edge in E, such that a node V_k is reachable from V_i, then we say that V_k is directly related to V_i. If V_k is reachable from V_i by traversing more than one edge in E, then we say that V_k is indirectly related to V_i.

From the definitions above, we can see that SAG has the following properties which are owned by iDM or AVG:

1. SAG is a logical graph. Nodes in SAG contain a sequence of components that ex-press structured, semi-structured, and unstructured pieces of the data, and all parts of the SAG can be computed lazily.
2. SAG may represent fine-grained logical entities in software artifacts. These entities can be files or structural elements inside files, such as a class, a function of source code, or a chapter, a section of a text document. SAG may also represent the outside structural information of files such as folder hierarchies. Furthermore, it is important to describe the relative order among software artifacts (such as the relative order of sections and chapters in the document), and we may represent them in the sequence B.
3. It is important to note that SAG is a very loosely structured model which does not enforce a schema over the data, that is, the set of attributes of each node may be different, and E may be empty set. In general, E contains explicit connections among nodes. When all the connections among nodes are not explicit, $E = \emptyset$.

In the following, we show detailed representation of some software artifacts present in Figure 1 a).



```
import java.net.*;
public class search{
    public static void main(String[] args) {
        try {
          InetAddress address=InetAddress.getByName…
         System.out.printIn(address);
         }
        catch (Exception e1) {
          System.out.printIn ("Sorry, the site or IP has …
```

(a) Software Artifacts (b) SAG

Figure 1. SAG represents software artifacts

We represent the "project1" folder as a node V_{project1} as follows:

- $V_{project1}$:

$$\begin{aligned}
\eta_{\text{project1}} &= \text{“project1”};\\
\tau_{\text{project1}} &= \{(\text{type, “folder”}), (\text{size}, 29\,684), (\text{createddate}, 21/07/2008\ 13{:}04),\\
&\quad (\text{updateddate}, 15/12/2008\ 18{:}45)\ldots\};\\
\chi_{\text{project1}} &= \langle\rangle; \qquad // \text{ the null value is denoted by } \langle\rangle\\
E_{\text{project1}} &= \{(A, L_A), (B, L_B)\}\\
&= \{(A, L_A) = ((V_{\text{software-design.doc}}, \text{“hasFile”}),\\
&\quad (V_{\text{search.java}}, \text{“hasFile”})), (B, L_B) = \langle\rangle\},
\end{aligned}$$

where L_A and L_B are the labels of the two sets of edges, respectively.

We represent the document "software-design.doc" as a node $V_{\text{software-design.doc}}$, and the components of $V_{\text{software-design.doc}}$ and its related nodes are partly shown as follows:

- $V_{\text{software-design.doc}}$:

$$\begin{aligned}
\eta_{\text{software-design.doc}} &= \text{“software-design.doc”};\\
\tau_{\text{software-design.doc}} &= \{(\text{type, “Docfile”}), (\text{size}, 3\,508),\\
&\quad (\text{createddate}, 21/09/2008\ 10{:}34),\\
&\quad (\text{updateddate}, 25/10/2008\ 11{:}42), (\text{author, “MingLi”}),\\
&\quad (\text{version}\ldots)\ldots\};\\
E_{\text{software-design.doc}} &= \{(A, L_A) = \langle\rangle, (B, L_B) = ((V_{\text{Introduction}}, \text{“hasSection”}),\\
&\quad (V_{\text{DatabaseDescriptions}}, \text{“hasSection”}),\\
&\quad (V_{\text{FunctionDescriptions}}, \text{“hasSection”})\ldots)\},
\end{aligned}$$

where B represents the relative order of sections in the document "software-design.doc".

- $V_{\text{Introduction}}$:

$$\begin{aligned}
\eta_{\text{Introduction}} &= \text{“Introduction”};\\
\tau_{\text{Introduction}} &= \{(\text{class, “section”}), (\text{createddate}, 21/09/2008\ 10{:}34),\\
&\quad (\text{updateddate}, 25/10/2008\ 11{:}42), (\text{author, “MingLi”}),\\
&\quad (\text{version}\ldots)\ldots\};\\
\chi_{\text{Introduction}} &= \langle\text{text content}\rangle.
\end{aligned}$$

We represent the file "search.java" as a node $V_{search.java}$, and the components of $V_{search.java}$ and its related nodes are partly shown as follows:

- $V_{\text{search.java}}$:

$$\begin{aligned}\eta_{\text{search.java}} &= \text{“search.java”};\\ \tau_{\text{search.java}} &= \{(\text{type}, \text{“Javafile”}), (\text{size}, 5\,786), (\text{createddate}, 27/08/2008\ 19{:}24),\\ &\quad (\text{updateddate}, 21/10/2008\ 09{:}52), (\text{author}, \text{“MingLi”})\ldots\};\\ E_{\text{search.java}} &= \{(A, L_A) = \langle\rangle, (B, L_B) = ((V_{\text{import}}, \text{“importDeclaration”}),\\ &\quad (V_{\text{search}}, \text{“classDeclaration”}))\}.\end{aligned}$$

- V_{import}:

$$\begin{aligned}\eta_{\text{import}} &= \text{“import”};\\ \tau_{\text{import}} &= \{(\text{package name}, \text{“java.net.*”}), \ldots\};\end{aligned}$$

The data and their relationships are represented as SAG that is shown in Figure 1 b). Each node is labeled with its name component. For the sake of readability, we have omitted all edge labels and the relative order among nodes.

There are many XML-based representations for source code (e.g. C, C++ and Java [1, 15]), and our data model can represent XML document in the way shown as follows: Each node in the XML document is represented as a corresponding node in SAG, and the connections among nodes are represented by E components.

4 QUERYING SAG ON TOP OF AVG

In this section, we describe how to query SAG on top of AVG. The key idea of our strategy is as follows. We encode SAG as AVG, and implement a query q_S on SAG by translating q_S to an equivalent query q_A over AVG. The approach is shown in Figure 2. The basic steps of the approach are:

1. Converting SAG to AVG;
2. Translating a query q_S on SAG to a query q_A on the AVG;
3. Getting the results from q_A, and converting them from AVG back to SAG.

In the following sections, we specify the steps listed above.

4.1 Transformations Between SAG and AVG

We show the transformations between SAG and AVG as follows:

1. SAG: name component $\eta_i \Leftrightarrow$ AVG: $(name, \eta_i)$
2. SAG: content component $\chi_i \Leftrightarrow$ AVG: $(content, text)$
3. SAG: attribute-value component $\tau_i = (W, T) \Leftrightarrow$ AVG: (W, T)

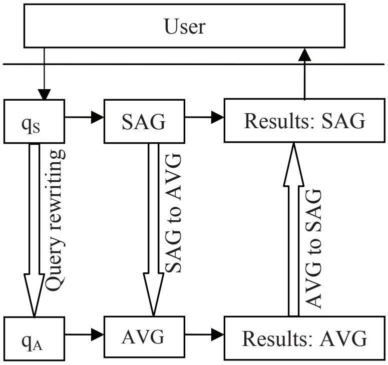


Figure 2. SAG query processing on top of AVG

4. SAG: E component

$$E_{Vi} = \{(A, B), L)\} \Leftrightarrow AVG : \begin{cases} \text{edges: } \{(V_i, V_{A1}, L_{A1}), \ldots, (V_i, V_{Am}, L_{Am})\} \cup \\ \{(V_i, V_{B1}, L_{B1}), \ldots, (V_i, V_{Bn}, L_{Bn})\} \\ \\ \text{attribute – value pair of } V_i : \\ (\text{SubNode}, [V_{B1}, \ldots, V_{Bn}]) \end{cases}$$

where $V_{Aj} \in A$ $(1 \leqslant j \leqslant m)$; $V_{Bk} \in B$ $(1 \leqslant k \leqslant n)$ and the value of SubNode is a sequence of nodes.

For example, the transformation between E component of $V_{\text{software}-\text{design.doc}}$ and AVG is shown as follows:

$$\begin{aligned} SAG \;:\; & E_{\text{software}-\text{design.doc}} = \{(A, L_A) = \langle\rangle, (B, L_B) = ((V_{\text{Introduction}}, \text{“hasSection”}), \\ & (V_{\text{DatabaseDescriptions}}, \text{“hasSection”}), (V_{\text{FunctionDescriptions}}, \text{“hasSection”}) \ldots)\}. \end{aligned}$$

$$\Leftrightarrow AVG : \begin{cases} \text{edges: } \{(V_{\text{software}-\text{design.doc}}, V_{\text{Introduction}}, \text{“hasSection”}), \\ (V_{\text{software}-\text{design.doc}}, V_{\text{DatabaseDescriptions}}, \text{“hasSection”}), \\ (V_{\text{software}-\text{design.doc}}, V_{\text{FunctionDescriptions}}, \text{“hasSection”}) \ldots\} \\ \\ \text{attribute – value pair of } V_{\text{software}-\text{design.doc}} : \\ (\text{SubNode}, [V_{\text{Introduction}}, V_{\text{DatabaseDescriptions}}, V_{\text{FunctionDescriptions}}]) \end{cases}$$

4.2 Translating q_S to q_A

The language used to query AVG model is a subset of iQL [11], which is similar in spirit to NEXI [16]. However, iQL includes features important for a DSMS, such as support for updates and continuous queries.

We present some example queries as follows:

- Q_1 : "software design"
  returns those nodes containing the phrase "software design".
- Q_2 : // Introduction / *["database"]
  returns those nodes that are directly related to a node named "Introduction". In addition, all returned results contain the keyword "database".
- Q_3 : // Project //Introduction [updateddate < yesterday ( )]
  returns those nodes named "Introduction" that are indirectly related to a node named "Project". In addition, all returned results have an updated date before yesterday.

Since SAG can be translated to AVG, a query q_S on SAG can be expressed as a equivalent query q_A on AVG.

Example 1. A user wants to find the sections which not only contain the keyword "search" but also pertain to the directory named "project1".

The corresponding query q_S is: 'return those nodes that are indirectly related to a node V_1 having η_1 = "project1" and τ_1 = (type, "folder"). In addition, all returned nodes contain the keyword "search"'.

Then q_S is equivalent to q_A: 'return those nodes that are indirectly related to a node having a set of attribute – value pairs {(name, "project1"), (type, "folder")}. In addition, all returned results contain the keyword "search"'. That is,

$$q_A := // \text{ project1[type = "folder"] // *["search"]}.$$

The returned nodes are $V_{\text{search.java}}$, $V_{\text{searchFunction}}$ and V_{search}.

In fact, a query q_S can be translated to a equivalent query q_A by transforming the SAG components into the AVG components, according to the transformation method discussed in Section 4.1.

5 ADDING SEMANTICS TO QUERY PROCESSING

iTrail is a semantic trail which provides a mapping from one query to another; it may be used to encode schema information from different data sources, without requiring full integration of all sources from the start. iTrail is defined as follows [12].

Definition 3 (iTrail)**.** iTrail could be unidirectional or bidirectional; an iTrail is denoted either as:

$$\psi := Q_1 \longrightarrow Q_2 \text{ or } \psi := Q_1 \longleftrightarrow Q_2.$$

The unidirectional trail states that Q_1 induces Q_2, i.e., whenever we query for Q_1, we should also query for Q_2. The bidirectional trail also states that Q_2 induces Q_1.

Users may define iTrails by themselves, or obtain a set of trail definitions by mining semi-automatically from content. For example, ontologies and Wordnet can be exploited to extract equivalences among keyword queries (e.g., $\psi := \mathit{car} \longrightarrow \mathit{automobile}$ could be automatically generated from Wordnet). Machine learning techniques can also be used to create keyword-to-keyword trails.

iTrail definitions are explored during query processing to improve the precision and recall of query results. In the following, we present how to add semantics to query processing through iTrails.

Example 2. Adding semantics to software artifacts query

The iTrails are assumed to be defined as follows:

$$\begin{aligned} \psi_1 &:= \text{search} \longrightarrow \text{query}, \\ \psi_2 &:= \text{function} \longrightarrow \text{class} = \text{function}. \end{aligned}$$

ψ_1 states that a query Q_1 := "search" induces a query Q_2 := "query", ψ_2 states that a query Q_1 := "function" induces a query Q_2 := [class = "function"].

When users query for keywords "search" and "function", the search should include not only the results of the original query Q := "search" and "function", but also the results of the query Q' := "query"[class = "function"].

6 EVALUATION

In this section, experiments have been performed:

1. to evaluate the performance of software artifacts query approach mentioned in Section 4;
2. to estimate how the performance of artifacts query would be affected by the semantics addition approach given in Section 5.

6.1 Experimental Setup

In order to evaluate our approach, we acquired software artifacts from the SCR directory of FreeMarker 2.1.5[2], which is a template engine to generate text and/or HTML. The artifacts contain 314 files, 18 folders, and the total size is 1.43 MB; the file types include XML, HTML, TXT, JAVA, etc. All experiments were performed on top of iMeMex system which we have extended to support software artifacts management, and the computer used for the experiments was a dual Intel® Atom™ CPU 1.66 GHz with 1 GB of RAM. SAG was constructed automatically by extracting these artifacts, which contained 556 nodes and 545 edges.

[2] http://freemarker.sourceforge.net/

6.2 Performance of Software Artifacts Query

The queries evaluated are based on both keywords and structure. The queries and their average response times are shown in Table 1. The response times are obtained on a warm cache, i.e., each query is run several times until the deviation on the average response time becomes small. From the experimental result, we can see that all queries response times are less than 0.5 seconds. In addition, SAG is essentially a graph model, and we could use the strategies of keyword query on graph [17, 18] to accelerate the query-response times. Thus, we can draw the conclusion that our approach is feasible according to [19], which suggested that a response time of less than 1 second would be available for every computer system.

Query ID	Expression	# of Results	Response time (s)
Q_1	html	292	0.48
Q_2	//testcase/*[“html”]	138	0.35
Q_3	date > 28.01.2003 size > 34 077	0	0.04
Q_4	test main	40	0.16
Q_5	animal class = “file”	6	0.19

Table 1. The number of results and the response times of queries

6.3 Performance of Semantics Addition

We compare the artifacts query without semantics addition to that with semantics addition by rewriting the queries utilizing iTrails technique.

We have assessed the results using the IR metrics recall and precision. Recall computes the percentage of the number of relevant retrieved documents for a query over the total number of relevant documents for that query. Precision computes the percentage of the number of relevant retrieved documents over the total number of retrieved documents.

iTrails used for evaluation are defined as follows:

$$\begin{aligned}
\psi_1 &:= \text{html} \longrightarrow //^*.\text{html} \cup //^*.\text{htm},\\
\psi_2 &:= \text{date} \longleftrightarrow \text{lastmodified},\\
\psi_3 &:= \text{test} \longrightarrow //\text{testcase}//^*,\\
\psi_4 &:= \text{animal} \longrightarrow //\text{elephant},\\
\psi_5 &:= \text{animal} \longrightarrow //\text{fish}.
\end{aligned}$$

The average response times of queries with semantic addition are shown in Table 2. Taken as a whole, the gap of response times between semantic addition and without semantic addition (see Table 1) is very limited. Furthermore, according to [12], the query response times are also in the acceptable range as the number and complexity of iTrails increase, so we can draw the conclusion that our approaches do not add much overhead for semantic addition.

Query ID	With iTrails applied	# of Results	Response time (s)
Q_1	ψ_1	301	0.48
Q_2	ψ_1	147	0.38
Q_3	ψ_2	5	0.05
Q_4	ψ_3	42	0.17
Q_5	ψ_4, ψ_5	12	0.19

Table 2. The number of results and the response times of queries with semantics addition

Figure 3 shows the recall and precision of queries with/without semantics addition. As we may notice, the recall or precision of all queries are improved in different degree by adding trails in a pay-as-you-go fashion. For example, the precision of Q_4 with iTrails (i.e., ψ_3) is about 5% higher than that without iTrails, and the recall of Q_5 with iTrails (i.e., ψ_4 and ψ_5) is about 50% higher than that without iTrails. Moreover, both the recall and the precision of Q_3 without iTrails are 0, because the attribute "lastmodified" rather than "date" is used to denote the information of the file date and time in these artifacts. After adding semantics, both the recall and the precision of Q_3 with iTrails (i.e., ψ_2) are sharply improved.

In summary, the experimental results show that our semantics addition method strongly improves the quality of query results when compared to the approach providing keyword and structural search which has no integration semantics of the data.

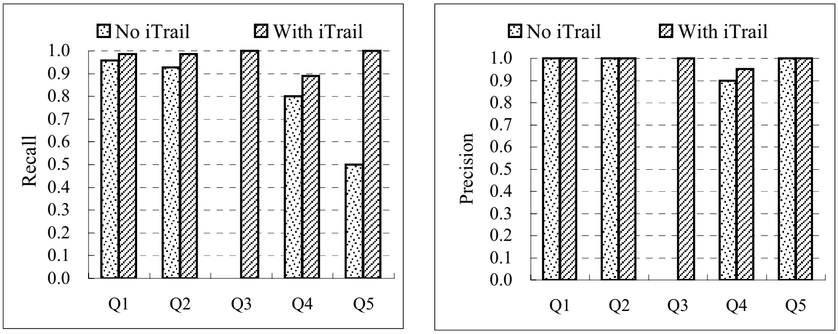


Figure 3. The recall and precision of queries with/without semantics addition

7 RELATED WORK

In this section, we compare our work with traditional work from several aspects: software artifacts descriptions and retrieval, as well as semantics addition. The comparison results are shown in Table 3.

Measurements	Our Work	Traditional Work	PAYG Features in Our Work
Software Artifacts Descriptions and Retrieval			
Software Artifacts Descriptions	SAG: loosely modeled	XML, graph, ontology, . . . : strict models	• without requesting to fully materialize SAG beforehand • with little up-front cost
Software Artifacts Retrieval	• query SAG by iQL • keyword-based query	• keyword-based query • ontology queries	• compute SAG lazily on demand • provide best-effort services in a PAYG fashion
Semantics Addition	• add semantics to query processing by iTrails • gradually enhance semantic query with little up-front cost	• utilize ontologies to represent artifacts and background knowledge • powerful semantic query with high up-front cost	add semantics to query processing gradually

Table 3. Comparison between our work and traditional work

7.1 Software Artifacts Descriptions and Retrieval

The different data models have been proposed to represent software artifacts for managing software artifacts effectively. XML-based source code representation has received considerable attention. In [15], a XML-based representation, JavaML, was presented to represent Java source code. A program representation approach which is based on language domain models and the XML markup language was proposed in [20]. srcML, an XML application, was provided to add structural information to unstructured source code text files, and srcML documents could be easily searched and queried with standard XML tools [21]. Utilizing an XML representation, the authors of [1] provided a tool platform to manage the fine-grained information about Java source code. In [2] a tool was described to support the transformation of software programs from ASCII plain text format to XML. Some people also represented software artifacts by graph model. In [3] conceptual graphs were provided to represent the code, and then a retrieval model was designed on these graphs.

In [22] Abstract Semantic Graph was presented to represent source code text. The model was composed of nodes and edges, where nodes represented source code entities, and edges represented relations. Both the nodes and the edges were typed and had their own annotations that denoted semantic properties. Some other people proposed the ontological model of software artifacts. In [4] authors provided ontological representations for both source code and document artifacts, ontology queries and Description Logic reasoning can be applied on these representations. In order to assist the communication between software developers for bug resolution, the authors of [23] presented an ontology to describe software, developers, and bugs. This ontology can be semi-automatically populated from existing artifacts.

All of these previous data models are based on a schema-first modeling strategy that needs mapping data to a schema or a domain model (e.g., ontology) before supporting query on software artifacts. The schema-first approaches make it hard to integrate information in a PAYG fashion. In our work, SAG is a loosely structured data model, thus it is more easy to describe software artifacts. In contrast to the traditional approaches which are usually tied to having a physical representation of the whole data before querying may be carried out, we do not require SAG materialized beforehand. That is, all nodes and edges in SAG may be computed lazily on demand. Moreover, the traditional approaches have been associated with high-cost, high-quality functionalities. However, we provide gradually enhanced services according to the amount of effort investment.

Note that incremental maintenance of software artifacts is somewhat similar in spirit to our work. In [24, 25] authors presented efficient incremental validation techniques for XML documents. In [26, 27] the ways were proposed to maintain the consistency among software artifacts during development. In [28] authors presented a formal model of incremental consistency checking for pervasive contexts. These approaches aim to maintain the database incrementally, as well as update and check the set of constraints incrementally. Both these approaches and our work focus on integrating or managing data gradually. In addition, these approaches are helpful for us to detect changes on software artifacts. However, the ways to update data in these approaches are still based on the schema-first modeling strategy.

7.2 Semantics Addition

Extensive efforts have been devoted to applying Semantic Web technologies in software engineering. Some researchers provided ontological representations for software artifacts (see Section 7.1), while others enhanced semantic software artifacts descriptions to support specific task by utilizing ontologies to represent software background knowledge (e.g., domain knowledge and design knowledge) [5, 29, 30]. Unlike the previous semantics-based approaches which require significant efforts to declare data semantics through complex ontologies to enrich query processing, we add semantics by iTrail. Our approach is much more lightweight, allowing users to add semantics over time in a PAYG fashion. In other words, our work gradually enhances semantic query with little up-front cost, while the traditional work supports the powerful semantic query with hard up-front cost.

8 CONCLUSIONS AND FUTURE WORK

Dataspace system provides an environment which could combine other techniques (such as ontology and IR techniques) for software artifacts management in a PAYG fashion. However, to the best of our knowledge, dataspace techniques have not gained any attention among researchers in the field of software engineering until now. In this paper, we present a strategy based on dataspace techniques to manage software artifacts in a PAYG fashion, and show how to extend and adapt the existing dataspace techniques to meet this goal. In the future, we plan to focus on the question of how to create traceability among software artifacts, and to realize the recovery of traceability links in a PAYG fashion.

Acknowledgements

This work was supported by National Nature Science Foundation of China (No. 609-70044, No. 60970044, No. 61272067 and No. 61363074), Natural Science Foundation of Guangdong Province of China (No. 7003721), and Guangxi Natural Science Foundation (No. 2013GXNSFAA019346).

REFERENCES

[1] Maruyama, K.—Yamamoto, S.: A CASE Tool Platform Using an xml Representation of Java Source Code. In Proceedings of the 4th IEEE International Workshop on Source Code Analysis and Manipulation 2004, pp. 158–167.

[2] McArthur, G.—Mylopoulos, J.—Ng, S. K. K: An Extensible Tool for Source Code Representation Using XML. In Proceedings of the 9th Working Conference on Reverse Engineering, IEEE Computer Society 2002, pp. 199–208.

[3] Mishne, G.—de Rijke, M.—Marx, M.: Source Code Retrieval Using Conceptual Graphs. Proceedings of RIAO, Citeseer 2004.

[4] Witte, R.—Zhang, Y.—Rilling, J.: Empowering Software Maintainers with Semantic Web Technologies. Proceedings of the 4th European Conference on the Semantic Web: Research and Applications, Vol. 4519, 2007, pp. 37–52.

[5] Wongthongtham, P.—Chang, E.—Dillon T.—Sommerville, I.: Development of a Software Engineering Ontology for Multisite Software Development. IEEE Transactions on Knowledge and Data Engineering, Vol. 21, 2009, pp. 1205–1217.

[6] Franklin, M.—Halevy, A.—Maier, D.: From Databases to Dataspaces: A New Abstraction for Information Management. ACM SIGMOD Record, Vol. 34, 2005, pp. 27–33.

[7] Ying, P.—Yong, T.—Xiaoping, Y.: Software Artifacts Management Based on Dataspace. Proceedings of the WASE International Conference on Information Engineering, IEEE 2009, pp. 214p–217.

[8] Halevy, A.—Franklin, M.—Maier, D.: Principles of Dataspace Systems. Proceedings of the 25th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, ACM 2006, pp. 1–9.

[9] Halevy, A. Y.: User-Focused Database Management. Proceedings of the 13th International Conference on Intelligent User Interfaces, ACM 2009, pp. 5–6.

[10] Dittrich, J. P.: iMeMex: A Platform for Personal Dataspace Management. Proceedings of SIGIR PIM Workshop, ACM 2006, pp. 40–43.

[11] Dittrich, J. P.—Salles, M. A. V.: iDM: A Unified and Versatile Data Model for Personal Dataspace Management. Proceedings of the 32nd International Conference on Very Large Data Bases, VLDB Endowment 2006, pp. 367–378.

[12] Salles, V.—Antonio, M.—Dittrich, J. P.—Karakashian, S. K.—Girard, R.—Blunschi, L.: iTrails: Pay-As-You-Go Information Integration in Dataspaces. Proceedings of the 33rd International Conference on Very Large Data Bases, VLDB Endowment 2007, pp. 663–674.

[13] Salles, M. A. V.—Dittrich, J.—Blunschi, L.: Intensional Associations in Dataspaces. Proceedings of the 26th International Conference on Data Engineering, IEEE Computer Society 2010, pp. 984–987.

[14] Fletcher, G. H. L.—van den Bussche, J.—van Gucht, D.—Vansummeren, S.: Towards a Theory of Search Queries. Proceedings of the 12th International Conference on Database Theory, ACM 2009, pp. 201–211.

[15] Badros, G. J.: JavaML: A Markup Language for Java Source Code. Computer Networks, Vol. 33, 2000, pp. 159–177.

[16] Trotman, A.—Sigurbjörnsson, B.: Narrowed Extended Xpath I (NEXI). Advances in XML Information Retrieval 2005, pp. 16–40.

[17] Kacholia, V.—Pandit, S.—Chakrabarti, S.—Sudarshan, S.—Desai, R.—Karambelkar, H.: Bidirectional Expansion for Keyword Search on Graph Databases. Proceedings of the 31st International Conference on Very Large Data Bases, VLDB Endowment 2005, pp. 505–516.

[18] He, H.—Wang, H.—Yang, J.—Yu, P. S.: BLINKS: Ranked Keyword Searches on Graphs. Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data, ACM 2007, pp. 305–316.

[19] Shneiderman, B.: Response Time and Display Rate in Human Performance with Computers. ACM Computing Surveys (CSUR), Vol. 16, 1984, pp. 265–285.

[20] Mamas, E.—Kontogiannis, K.: Towards Portable Source Code Representations Using XML. Proceedings of WCRE, Citeseer 2000, pp. 172–182.

[21] Maletic, J. I.—Collard, M. L.—Marcus, A.: Source Code Files as Structured Documents. Proceedings of the 10th InternationalWorkshop on Program Comprehension, IEEE Computer Society 2002, pp. 289–292.

[22] Devanbu, P. T.—Rosenblum, D. S.—Wolf, A. L.: Generating Testing and Analysis Tools with ARIA. ACM Transactions on Software Engineering and Methodology, Vol. 5, 1996, pp. 42–62.

[23] Ankolekar, A.—Sycara, K.—Herbsleb, J.—Kraut, R.—Welty, C.: Supporting Online Problem-Solving Communities with the Semantic Web. Proceedings of the 15th International Conference on World Wide Web, ACM 2006, pp. 575–584.

[24] Papakonstantinou, Y.—Vianu, V.: Incremental Validation of XML Documents. Proceedings of the 9th International Conference on Database Theory, Springer 2003, pp. 47–63.

[25] Barbosa, D.—Mendelzon, A. O.—Libkin, L.—Mignet, L.—Arenas, M.: Efficient Incremental Validation of XML Documents. Proceedings of the 20th International Conference on Data Engineering, IEEE Computer Society 2004, pp. 671–682.

[26] Nentwich, C.—Capra, L.—Emmerich, W.—Finkelstein, A.: xlinkit: A Consistency Checking and Smart Link Generation Service. ACM Transactions on Internet Technology (TOIT), Vol. 2, 2002, pp. 151–185.

[27] Reiss, S. P.: Incremental Maintenance of Software Artifacts. IEEE Transactions on Software Engineering, Vol. 32, 2006, pp. 682–697.

[28] Xu, C.—Cheung, S. C.—Chan, W. K.: Incremental Consistency Checking for Pervasive Context. Proceedings of the 28th International Conference on Software Engineering, ACM 2006, pp. 292–301.

[29] Zhao, Y.—Dong, J.—Peng, T.: Ontology Classification for Semantic-Web-Based Software Engineering. IEEE Transactions on Services Computing, 2009, pp. 303–317.

[30] Barla, M.—Tvarožek, M.—Bieliková, M.: Rule-Based User Characteristics Acquisition from Logs with Semantics for Personalized Web-Based Systems. Computing and Informatics, Vol. 28, 2009, pp. 399–427.

Ying Pan received the M. Sc. degree in computer science from Huazhong University of Science and Technology in 2006, and the Ph. D. degree from Department of Computer Science, Sun Yat-sen University in 2011. She is now an Associate Professor in College of Computer and Information Engineering, Guangxi Teachers Education University, China. Her current research interests include dataspace and knowledge engineering.

Yong Tang (corresponding author) received the M. Sc. degree in computer software from Wuhan University in 1990, and the Ph. D. degree in computer software and theory from University of Science and Technology of Beijing (China) in 2001. He is now a Professor and Ph. D. supervisor in School of Computer Science, South China Normal University, Guangzhou, China. His main research interests include database and knowledge engineering.