Inferring the Data Access from the Clients of Generic APIs

Hui Song, Gang Huang, Yingfei Xiong, Yanchun Sun

Inferring the Data Access from the Clients of Generic APIs

Hui Song*†, Gang Huang*[1], Yingfei Xiong*, Yanchun Sun*

*Key Lab of High Confidence Software Technologies (Ministry of Education)
School of Electronic Engineering & Computer Science, Peking University, China
†Lero: The Irish Software Engineering Research Centre, SCSS, Trinity College Dublin, Ireland
Email: hui.song@scss.tcd.ie, {huanggang, xiongyf04,sunyc}@sei.pku.edu.cn

Abstract—Many programs access external data sources through generic APIs. The class hierarchy of such a generic API does not reflect the schema of any particular data source, and thus it is hard to clarify what data an API client accesses and how it obtains them. This makes it difficult to maintain the API clients. In this paper, we show that the data access of an API client can be recovered through static analysis on the client's source code. We provide a formal and intuitive way to represent the data access, as a graph of so-called summoning snippets. Each snippet stands for a type of data accessed by the client, and carries the code slice from the client about how to obtain the data via the API. We provide an automated approach to inferring a complete and well-simplified set of summoning snippets from the client source code, based on points-to analysis and code slicing. We implement this approach as a development assistant tool, and evaluate it on eight open source data processing programs, with average precision and recall of 89% and 95%, respectively. Further inspection of these clients, as well as a user study about writing data accessing code on their data sources, show that the inference results are useful in the inspection of existing clients and the development of new data access logics.

I. INTRODUCTION

Many computer programs access external sources, such as parsing XML documents, monitoring the runtime system states, etc. Dedicated APIs are developed for different types of data sources, such as the DOM API for XML files and the JMX API for Java systems. Accordingly, we call the programs *API clients*. A data source usually provides more than one type of data element, and the client retrieves different types of data by invoking the APIs in different ways. For example, an XML file describing a file system may provide two types of data: directories and files. A client may first obtain a root directory, then query sub directories and files recursively, and then obtain the attributes of each file.

Many software maintenance tasks require the understanding of the *data access* of an API client, i.e., what types of data a client retrieves, and how it invokes the API to retrieve the data. First, we often need to inspect the existing data accessing logics in the API clients. For example, when the data source updates, we would like to know if the evolution on the data schema will cause the existing client to be incompatible. For another example, given a transformation program that converts a data source into other formats, we may want to check if all data types provided by the source are fully covered by the client. To answer these questions, we need to know what data are accessed by the client. Second, when evolving the client to add new functions, or developing a new client from scratch, we need to write new data accessing code. To do this, we need to know what data are provided by the source and how to retrieve them by invoking the API. While the documents of data sources are often missing or out of date [1], the source code of existing clients is always a potential source of such information, and we can use their existing data access as a reference to write our code.

On the other hand, many data accessing APIs are generic, which makes it difficult for developers to understand the data access from reading the client code. A generic API is designed for many data sources, and thus its class hierarchy does not reflect the data schema of any particular source. For example, all XML elements in the DOM API, regardless of the types, are abstracted as an `Element` class, and any attribute is obtained by calling `getAttribute`. As a result, to know what data a statement retrieves, we cannot just refer to the static classes of the result and argument variables, or the method called, but have to trace the dependency from this statement to other ones. Consider this sample code on the XML file system example.

```
Element e = a.getElementsByTagName(t).item(0);
String n = e.getAttribute("name");
```

We do not know whether the first line retrieves a file or a directory, unless we know the value of the variable `t`. And without knowing the data type of `e`, we do not know whether the second line retrieves a file name or a directory name. In summary, to know what data it accesses and how it obtains them, we need to find out the API invocations, as well as the value of the constants and the types of previously retrieved data elements, used by each of the invocations. As the scale of the code can be big and the API-related code is often tangled with the business logics, it can be very difficult to figure out all the data access by reading the client code.

In this paper we propose a novel approach to inferring and representing the data access of an API client. Our

---

[1]corresponding authors

contribution can be summarized as follows.

- We provide a new way to represent the data access of API clients. Any type of data elements that the client may access, along with the code slice in the client to obtain them, is abstracted as a *summoning snippet* (imaging a "summoning spell" to call the data from under the API). The graph of snippets intuitively describes what data the client accesses, and the composition of the code slices from snippets describes how to obtain the data from previously retrieved data.

- We provide an automated approach to inferring the data access, based on the static analysis of the client source code. In particular, we extend points-to analysis to trace the data dependency between API invocations, and deduce all the possible types of data that can be accessed by the client. Then we use backward slicing to extract the relevant code to obtain each type of data.

- We applied the approach on eight open source clients of three APIs, and inferred the summoning snippets with 89% precision and 95% recall, in average. The results revealed the proper configuration of points-to analysis for inferring data access. We inspected the clients with the inference results and discovered two problems in the clients. A user study showed how they can be used in the programming of data access logics.

The implementation and the experiments are all open sourced at http://code.google.com/p/smatrt.

The rest of the paper is structured as follows. Section 2 gives an overview of the approach. Section 3 defines the summoning snippets. Section 4 and 5 present the inference approach. Section 6 and 7 report the implement and the experiments. Section 8 discusses related work and Section 9 concludes the paper.

II. APPROACH OVERVIEW

This section gives a brief overview of our approach to inferring and representing the data access of API clients.

We use a made-up client of JMX (Java Management eXtension) API as a running example throughout this paper, as shown in Figure 1. The client first obtains an element for memory information (Line 5), and then uses it to retrieve the heap memory usage (7) and the verbose state (9). After that, it obtains the used or maximal heap size (11-13). Finally, it obtains the garbage collectors (14-15), and gets their collection counts (17). We can see that even for this tiny client, it is already difficult to figure out all the above data access logic just by reading the code.

We use a graph of *summoning snippets* to represent the data access of a client. Each snippet abstracts a unique way embedded in the client to obtain data from the API, recording the involved API invocation instruction, and the data elements or constant values used by this instruction. Figure 2 shows the snippets of Figure 1 in a graphical way. Each snippet corresponds to a data type discussed before. An

```
 1 public class JMXClient
 2   static MBeanServerConnection mbsc=null;
 3   public static void entry(MBeanServerConnection server){
 4     mbsc=server;
 5     ObjectName memory=ObjectName.getInstance("type=Memory");
 6     Wrap memwrap=new ONWrap(memory);
 7     Object heapUsage=memwrap.get("HeapMemoryUsage");
 8     Wrap huwrap=new CDSWrap(heapUsage);
 9     Boolean verbose=(Boolean)memwrap.get("'Verbose");
10     String attr="";
11     if(verbose.booleanValue()) attr="max";
12       else attr="used";
13     System.out.println(huwrap.get(attr));
14     Set gcs=mbsc.queryNames(
15       ObjectName.getInstance("*type=GarbageCollector"),null);
16     for(Iterator it=gcs.iterator(), int t=0; it.hasNext();)
17       t+=(Integer) ((new ONWrap(it.next())).get("CollectionCount");
18   }
19   interface Wrap{ Object get(String s);}
20   class ONWrap implements Wrap{
21     ObjectName on=null;
22     ONWrap(Object core){on=(ObjectName)core;}
23     Object get(String s){return mbsc.getAttribute(on,s);}
24   }
25   class CDSWrap implements Wrap{
26     CompositeDataSupport cds=null;
27     CDSWrap(Object core){cds=(CompositeDataSupport)core;}
28     Object get(String s){return cds.get(s);}
29   }
30 }
```

Figure 1.   Sample client code on JMX API: A running example

arrow $a \rightarrow b$ indicates that b depends on the data element obtained from a, while the snippets without incoming arrow only depends on constant values or the parameters of the entry method. We list the contents of three snippets as below.

```
$Memory = {ObjectName.getInstance("*type=Memory");}
$HeapMemoryUsage =
  {!server.getAttribute($Memory="HeapMemoryUsage");}
$CollectionCount =
  {!server.getAttribute($Gcs.iterator().next(),"CollectionCount")}
```

The first snippet means that to retrieve a memory element, the client invokes `getInstance` with a constant argument `"*type=Memory"`. The second snippet uses a *global input* (marked by "!", that is the parameter of the entry method) named `server`, and an element obtained from *another snippet* (marked by "$") named `Memory`. The third snippet also uses another snippet `Gcs`, but before using it as a parameter, it first invokes two library methods, `iterator` and `next` to process the data. We can compose snippets together by replacing the "$" identifiers with their associated snippets, getting draft code to retrieve the target type of data. It is worth noting that the summoning snippets are different from the code snippets focusing on the pure API usage, such as Jungloid [2] and XSnippet [3]. For example, Jungloid and XSnippet will deduce only one snippet for API usage, i.e., "from a `CompositeDataSupport` and a `String`, we can get an `Object`". However, as the argument values are different, we deduce different summoning snippets for
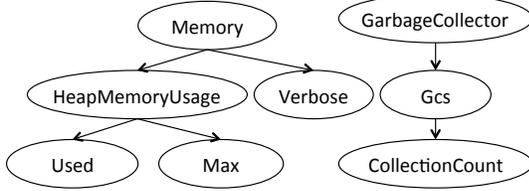
Figure 2. Graphical view of summoning snippets for the running example

different data types.

We provide an automated approach to inferring the summoning snippets. The approach takesthree inputs: the source code of the client, the namespace where all API invocations belong to, and the name of the entry method. The output is a complete and well-simplified set of summoning snippets from the client, as illustrated in Figure 2 and the sample contents above.

The inference is based on a simple heuristic: every API invocation with a return value retrieves a data element. Invocations to different methods retrieve different types of data. If the arguments passed to two invocations are different constants, or data elements of different types, the invocations return different types of data. Here the meaning of *API invocations* is general, including initializing an instance of an API class, calling its methods, and accessing its field.

Based on this heuristic, our approach has two steps. First, we analyze the client code to find the API invocations and all possible arguments passed to the invocations. For example, from Line 13 and Line 28 of Figure 1, the code analysis tells that the client invokes the `CompositeDataSupport.get` method, with argument `cds` that is the result of another snippet (from Line 7), and `s` as either `"max"` or `"used"`. Second, we perform a backward slicing [4] to construct the snippet for each API invocation with a different set of arguments. As we know `attr` in Line 13 could point to two different constants, the invocation in Line 28 leads to two snippets. For each snippet, we extract the necessary instructions in the client that provide the argument to this invocation, such as:

$Used =
 {(((CompositeDataSupport)$HeapMemoryUsage).get("used");}

We inline intermediate variables to get a compact representation.

The inference has several challenges. First, the argument of an invocation may be defined in another method or class, and to trace its source we need to consider the complex structures such as assignments, parameter passing, field access, virtual method binding, etc. Second, static code analysis usually requires the complete source code, but when analyzing the clients, we usually do not have the source code for APIs and third-party libraries. Third, an API invocation instruction may be launched in different execution paths (e.g., Line 23 can be launched from Line 7, 9 or 17), and

its argument may come from different sources (such as `s` in Line 28, whose source variable `attr` in Line 13 may be either `"max"` or `"used"`). Therefore, a single instruction may retrieve different types of data elements. Finally, slicing the snippet requires an execution path, and constructing all such paths is exhausting for static analysis.

To address these challenges, we adopt a points-to analysis [5] approach to trace the arguments of API invocations. We extend the basic approach by defining the API or library invocation results as a new kind of objects, to directly trace the data obtained from API, and also to handle the absence of API or library code. We choose a suitable context-sensitivity for points-to analysis to differentiate invocations in different execution paths, and define the rule to handle API invocations with multi-source arguments, in order to ensure that for each different way of API invocation, a corresponding object is constructed. Finally, we record the group of arguments used in each way of API invocation, and designed a slicing algorithm to locate the relevant instructions from the recorded argument objects and the def-relation [6] between instructions and objects, without constructing the full execution traces from scratch.

In the following we formally define summoning snippets, and then explain our inference approach in details.

III. THE SUMMONING SNIPPETS

DEFINITION 1. A **summoning snippet** ("snippet" for short) is a code slice of an API client that invokes the API to obtain a specific type of data elements. Formally speaking, a snippet s is a 4-tuple $s = (name, req, aux, seq)$. Here req is the set of other snippets that provide requisite results. If $s_1 \in s.req$, we also say that s **depends** on s_1. The set aux contains the entry method parameter and constant data used by the snippet as inputs, and seq is a sequence of instructions extracted from the client that obtain the target data from the input ones. The instructions include the API invocations and the necessary auxiliary instructions such as downcasting and the invocation to third party libraries. However, the invocation to client-defined classes or methods are not included. For the sake of clarity, we require each snippet reflect only one data type, and thus each seq contains one and only one API invocation instruction.

DEFINITION 2. Two snippets s_1, s_2 are **equal**, noted as $s_1 = s_2$, if and only if $s_1.req = s_2.req$, $s_1.aux = s_2.aux$, and $s_1.seq \cong s_2.seq$. Here the equivalence "$\cong$" between two instruction sequences means it is possible to make them identical by legally reordering the instructions. Intuitively, if two snippets are equal, they represent the same way to invoke the API.

DEFINITION 3. A snippet s_1 can be **composed** into another snippet s_2, noted as $xs = s_2 \propto s_1$. The result xs has the same structure as a summoning snippet, but does not satisfy the requirement of "containing only one API invocation", and we call it an extended snippet. If

s_2 does not depend on s_1, their composition is still s_2, otherwise the composition is calculated as follows. $xs.req = s_2.req - \{s_1\} \cup s_1.req$, $xs.aux = s_2.aux \cup s_1.aux$, and $xs.seq = s1.seq + s2.seq$. Intuitively, xs describes how to retrieve the same type of data as s_2, but it does not depend on s_1. Generally speaking, when we want to write the code to retrieve the type of data represented by s, but in the context of the current client (local variables, fields, etc.) there lacks some types of data element required by s, we can keep compositing other snippets into s to eliminate the requisitions, and finally got the code draft.

We visualize summoning snippets as a Directed Acyclic Graph (DAG), as shown in Figure 2. The vertices are the snippets, and the edges are dependencies (in a reverse direction, showing how the data flows). The DAG intuitively depicts the relations between the data types. Vertically, it shows if two data types have direct or indirect dependencies. Horizontally, it reveals how two types are close to each other, such as depending on the same third data type.

IV. SOURCE CODE ANALYSIS

This section presents the points-to analysis inferring all the different ways of API invocations in a client.

A. Background of points-to analysis

Points-to analysis is a static analysis technique. It simulates all the possible executions of source code, and constructs 1) a virtual *heap* storing all the objects produced by the code, and 2) a *points-to mapping* from each variable in the code to the objects it may point to. What to store in the heap is determined by the definition of objects, and usually includes the locally allocated instances and constant values.

The analysis can be illustrated in two steps. The first step is local analysis inside each method declaration. Take Line 9 in Figure 1 as an example. We create an object for the constant `"Verbose"`, and let the argument of `get` method point to it. In the meantime, we let the variable `verbose` point to a temporal object standing for the return value. Separately, in the declaration of `get` at Line 23, we let the argument of `getAttribute` point to the parameter s, and make the return variable point to the result of this method. In the second step, the local analysis results are merged together. Following the above example, by propagating arguments and return values, s points to `"Verbose"`, and the variable `verbose` points to the result of `getAttribute`. To handle polymorphism, this merging stage will use the real type of the receiver object to decide the method to merge.

The simplest and fastest analysis is to merge the same method declaration into every place it is invoked. However, such course-grained analysis is easy to produce fault points-to mappings. The solution is context-sensitivity, i.e., to make several copies of the same method declaration, and merge different copies into the invocation places that have different contexts. For object-oriented programs, two common contexts are the *receiver* which is the host object when an instance method is invoked [7], and the *callsite* which is the location where the method is invoked [5]. Researchers also proposed other finer contexts, such as a chain of receivers or call-sites, or the parameters, etc. Fine-grained context-sensitivity extremely increases the size of heap and points-to mappings, and decreases the analysis performance.

B. Points-to analysis for data access inference

To infer the data access, we define a new type of objects for points-to analysis and select the proper context.

Our objective is to record the results of API invocations, and find out how they are used in other invocations. Therefore, we directly define the API invocation results as objects. Defining the return values of API method calls and field accesses as objects is our extension to points-to analysis. It has the bonus advantage to avoid analyzing the API source code. We also extend the idea into third-party libraries, and regard the results of library invocations as objects, so that only the source code of the client itself is required.

Regarding the precision of analysis, we use the composition of single receiver and single call-site as the context. The call-site sensitivity is necessary when the client wraps the API method by its own one, such as `ONWrap.get` in Line 23. Without any context, we would merge the same method declaration into the two invocations to it at Line 7 and Line 9. Thus the variable s at Line 23 can be either `"HeapUsage"` or `"Verbose"`, and the return value can be used as either `Boolean` (Line 9) or `CompositeDataSupport` (Line 27). This will lead to an in-existing snippet such as

$Verbose = {(Boolean)!server
.getAttribute($Memory,"HeapMemoryUsage");}

Using call-site as a context, our conclusion will be refined into "if and only if `ONWrap.get` is invoked *at Line 9*, the parameter of `getAttribute` is `"Verbose"`, and the result is `Boolean`", avoiding the above snippet. The receiver-sensitivity is necessary when the client wraps the API classes by its own ones, such as `ONWrap` in Line 20. Without any context, from the invocation to `ONWrap.get` at Line 9 and Line 17, we see the variable s points to both `"Verbose"` and `"CollectionCount"`, and from the two class instantiations at Line 6 and Line 17, on points to the memory (Line 5) and the garbage collector (Line 15). Therefore, we get four compositions to invoke `getAttribute`, with two of them incorrect, such as

$Verbose = {(Boolean)!server
.getAttribute($CollectionCount,"Verbose");}

Using receiver as an context, the conclusion is narrowed to "if and only if `ONWrap.get` is invoked on the instance from Line 6 (whose on is memory), the argument s can be `"Verbose"`". Our experiments show that this context is enough for common data access clients.

C. The analysis in detail

Based on the above configuration and extension, we describe our points-to analysis approach as follows.

We abstract the source code into instructions, blocks and variables, based on the *Intermediate Representation* defined by WALA [8]. The left part of Figure 4 shows part of the abstraction for Figure 1. We name the instructions (rectangles) and variables (circles) after the line numbers and their orders of appearance. The variables come from left-values of instructions, parameters, and return values. After a branching block (b_3), a variable (such as $v_{12.2}$) is inserted to merge the homonymic variables from different branches. There are two relations between instructions and variables, *Use* and *Define*, as solid and dashed lines, respectively.

The analysis outputs a heap H storing four types of objects: `Input` for global inputs, `Const` for constants, `Alloc` for instances of client-defined classes, and `Obt` for the ones obtained from API or library invocations. The `Const` objects are determined only by its value. The `Alloc` objects are determined by both the allocation instructions and their contexts. For `Obt` objects, since invoking the same method with different arguments may obtain different types of data, we also include the argument values to differentiate `Obt` objects. Therefore, besides the instruction and the context, we also record a group `seed` objects, which are used by the instruction as the receiver and the arguments. The analysis also outputs a points-to relation $Pt \subseteq (V \times C \times H) \cup (H \times F \times H)$, indicating what objects a variable or a field of an object may point to, under a context $c \in C$ that is a tuple of one object and one instruction.

Figure 3 illustrates how we build H and Pt, in the form of denotational semantics on instructions.

Equation 1 means that after an assignment, the points-to relation is updated so that v_l points to the objects that are originally pointed by v_r. The semantics of static invocation (Equation 3) is a composition of three functions: propagating objects from the argument to the parameter, then executing the method declaration, and finally propagating the return value back. Here the propagation is a cross-context assignment defined in Equation 2, because the execution of m is under a new context with empty receiver and the callsite s. The instance method invocation is similar (Equation 4), except that if the receiver variable v_r points to multiple objects, we execute the method body multiple times, taking each object as the receiver. A `new` instruction (Equation 5) is a composition of an *allocation* that creates a new object (Equation 6), and an *invocation* to the constructor on the newly allocated object. Equation 7 defines the semantics of a branch: execute the two sub blocks separately and merge the objects pointed by the variables with the same name. We design the semantics based on the basic theory by Grove et al. [5] and the receiver sensitivity by Milanova et al. [7].

Equations 8-9 show how we construct `Obt` objects. For an
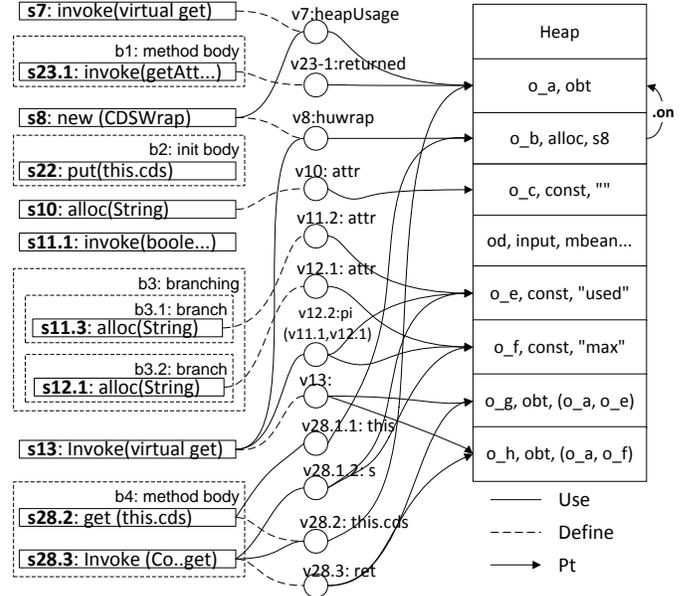

Figure 4.   Sample points-to analysis result

invocation to API or library method (the one not defined in the client), the receiver and parameter variables may point-to multiple objects, and thus the instruction may return different types of data. Recall the invocation to `get` in Line 28 ($s_{28.3}$ in Figure 4). There are two different ways to invoke `get`, from o_a and o_e, or from o_a and o_f, obtaining the used or maximal heap size, respectively. To ensure the different results are represented by different objects, we calculate the Cartesian product of these object sets pointed by the receivers and arguments, and for each tuple e_k in this product, we construct a new object o_k. The tuple e_k is recorded as the seed of o_k. The semantics of API class allocation and static method invocations are similar, except that there is no receiver. Equation 9 shows that if the method is client-defined, but some of the receiver objects are in type of `Obt`, we also construct `Obt` objects to represent the invocation results, as shown in Equation 9. This rule is also applicable to downcasting.

Since the constant values are meaningful in the invocation of API methods, we calculate their values during points-to analysis (Equation 10). For a binary operation op, if its two operand all point to constant objects, we calculate the value for each of the compositions between the two object sets, and create a new constant object from the calculation result.

The analysis starts from an entry method, with the initial H of `Input` containing its parameters. It takes an iterative solving process until $Heap$ and Pt are all stable [8].

Figure 4 shows part of the points-to analysis result. For the sake of simplicity, we omit some variables and all the contexts. The right most rectangles stand for the objects. We display their IDs, types, values(for `Const`), and seeds (for

$$\llbracket s : v_l = v_r \rrbracket_c \;=\; \langle H, Pt \rangle \to \langle H, Pt \cup (v_l, c) \to Pt^c(v_r) \rangle; \tag{1}$$

$$\llbracket v_1 := v_2 \rrbracket_{c_1,c_2} \;=\; \langle H, Pt \rangle \to \langle H, Pt \cup (v_1, c_1) \to Pt^{c_2}(v_2) \rangle \tag{2}$$

$$\llbracket s : v_l = \mathrm{T}.m(v) \rrbracket_c \;=\; \llbracket v_l := r \rrbracket_{c,(\epsilon,s)} \circ \llbracket m \rrbracket_{\langle \epsilon, s \rangle} \circ \llbracket p := v \rrbracket_{(\epsilon,s),c}; \tag{3}$$
$$(m \in \texttt{Clnt}, p = \texttt{fp}(m), r = \texttt{ret}(m))$$

$$\llbracket s : v_l = v_r.m(v) \rrbracket_c \;=\; \bigcup_{o_k \in pt^c(v_r)} \llbracket v_l := r \rrbracket_{c, \langle o_k, s \rangle} \circ \llbracket m_k \rrbracket_{\langle o_k, s \rangle} \circ \llbracket p := v \rrbracket_{\langle o_k, s \rangle, c}; \tag{4}$$
$$(m_k = \texttt{vir}(o_k, m), m \in \texttt{Clnt}, p = \texttt{fp}(m), r = \texttt{ret}(m), o_k \notin \texttt{Obt})$$

$$\llbracket s : v_l = \mathbf{new} \ \mathrm{T}(v) \rrbracket_c \;=\; \llbracket v_1.\mathbf{init}(v) \rrbracket_c \circ \llbracket v_l = \mathbf{alloc} \ \mathrm{T} \rrbracket_c; (\mathrm{T} \in \texttt{Clnt}) \tag{5}$$

$$\llbracket s : v_l = \mathbf{alloc} \ T \rrbracket_c \;=\; \langle H, Pt \rangle \to \langle H \cup \{o\}, Pt \cup ((v_1, c) \to \{o\}) \rangle; (o = \texttt{alloc}(T, c.r, s)) \tag{6}$$

$$\llbracket b : \mathbf{if} \ b_1 \ \mathbf{el} \ b_2 \ \mathbf{fi}; \pi \rrbracket_c \;=\; \llbracket \pi = \pi.1 \rrbracket_c \circ \llbracket \pi = \pi.2 \rrbracket_c \circ (\llbracket b_1 \rrbracket_c \cup \llbracket b_2 \rrbracket_c); \tag{7}$$

$$\llbracket s : v_l = v_0.m(v_1, ... v_n) \rrbracket_c \;=\; \langle H, Pt \rangle \to \langle H, Pt \rangle \cup \bigcup_{e_k} \langle \{o_k\}, ((v_l, c) \to \{o_k\}) \rangle; \tag{8}$$
$$(m \notin \texttt{Clnt}, e_k \in Pt(v_0) \times ... \times Pt(v_n), o_k = \texttt{obt}(c, s), o_k.seed = e_k)$$

$$\llbracket s : v_l = v_0.m(v_1, ... v_n) \rrbracket_c \;=\; \langle H, Pt \rangle \to \langle H, Pt \rangle \cup \bigcup_{e_k} \langle \{o_k\}, (v_l, c) \to \{o_k\} \rangle; \tag{9}$$
$$(m \in \texttt{Clnt}, r = Pt(v_1) \cap \texttt{Obt}, e_k \in r \times ... \times Pt(v_n), o_k = \texttt{obt}(c, s), o_k.seed = e_k)$$

$$\llbracket s : v_l = \texttt{op}(v_1, v_2) \rrbracket_c \;=\; \langle H, Pt \rangle \to \langle H \cup O, Pt \cup ((v_l, c) \to O) \rangle; (pt^c(v_1) \cup pt^c(v_2) \subseteq \texttt{Const}) \tag{10}$$
$$O = \{\texttt{const}(\texttt{op}(\omega)) | \omega \in Pt^c(v_1) \times Pt^c(v_n)\}$$

Figure 3.   Selected and simplified semantics of points-to analysis

Obt). The first API invocation (s_7) creates an Obt object o_a, and the return of method ONWrap.get makes v_7 point to o_a. After that, s_8 creates an Alloc object o_b, and the following constructor invocation makes the field $o_b.m$ point to o_a. The instructions from s_{10} to s_{12-1} create three Const objects, and the branching block b_3 makes the $v_{12.2}$ point to both o_e and o_f. After that, the invocation to CDSWrap.get passes these two objects to the parameter $v_{28.1.2}$, and finally the API invocation CompositeData.get creates two Obt object o_g and o_h from the two sets of seed objects (o_a, o_e) and (o_a, o_f), respectively.

V. SUMMONING SNIPPET CONSTRUCTION

This section describes how to construct the summoning snippets from the result of source code analysis.

In the heap H, each API obtained object stands for one type of data accessed by the client. Therefore, in the first step, we construct a summoning snippet from each of them. Here we decide whether an Obt object is from API or library by comparing the accessed class with the API namespace lists provided by users. We construct the content of a snippet s from an API-obtained object o as follows. We first find the seed objects $seeds$ of o, i.e., the objects used as the arguments of the instruction that output o. For example, for object o_g, the seed is $\{o_a, o_e\}$. If there is an object $o' \in seed$ which is another API-obtained object (i.e., o_a), then we add the summoning snippet corresponding to o' into $s.req$. Second, if o' is a Const or Input (i.e., o_e), we add its value ("used") or the entry method parameter name into $s.aux$. Third, we put the instruction that define o as the last

member in $s.seq$. For this example, we get the snippet as follows.

$Used =
    {(((CompositeDataSupport)$HeapMemoryUsage).get("used");}

If there is an $o \in seeds$ which is a library obtained object, then it means a library invocation is required before obtaining the target data. Thus we put the instruction that defines o into the head of $s.seq$, and go on to handle $o.seeds$ following the above steps.

The fine-grained points-to analysis makes a lot of copies of variables, instructions and objects, and thus the step above may lead to a big number of equal snippets. We de-duplicate the resulting summoning snippets by iteratively find and merge the equal ones (DEFINITION 2 in Section III).

We assign an intuitive name to each snippet, by first combining the name of the invoked API method, the constant value used as its parameter, and the name of the variable that stores the invocation result. We eliminate the common prefixes or suffixes, and remove the repeated parts in a name.

VI. IMPLEMENTATION

We implement the approach as a programming assistant tool, with a graphical editor of the summoning snippets implemented upon the Eclipse Graphical Modeling Framework (GMF, [9]). A snapshot is shown in Figure 5. The central editor shows the DAG of the snippets. If a single snippet is selected, a pop-up window prints its content. If several snippets are selected, the window prints their composition. We implement points-to analysis by extending the IBM
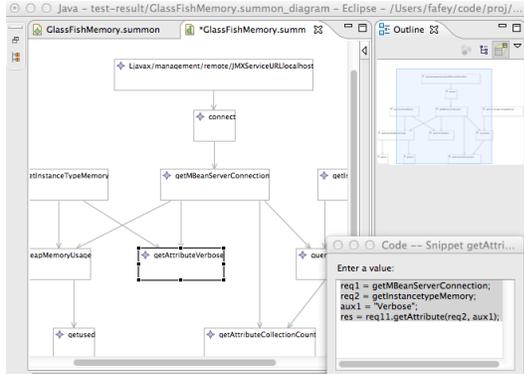
Figure 5.   Graphical tool of summoning snippets

WALA code analysis tool [8]. WALA, with "JDT CAst" extension, abstracts Java source code into an intermediate representation and performs different kinds of static analysis on it. We extend WALA by defining a new type of objects, designing new rules to create these objects, and adjusting existing solving rules to adapt to them. We also extend its default strategy to include receiver and callsite sensitivity.

VII. EVALUATION

We use experiments to answer the following questions. 1) Is the approach feasible on real API clients, and how are the inference results close to real data access of the clients? 2) How does the inferred summoning snippets assist the inspection of existing data access logics? 3) How do developers use them when writing new data access logics? To answer these questions, we applied the approach on eight open source API clients, used the results to inspect the subject clients, and performed a user study to see how other developers use them when developing.

A. Inference experiments

We selected eight open source clients on three different APIs. The first four clients use JMX API to retrieve the runtime state of one database system and two JEE systems, respectively. The next three clients use DOM API to access XML format data about the photo information from flickr, the bibliography source used by MS Office, and the log of an instant message system. The last client analyzes the JSON files of a social network aggregation system FriendFeeder. The first four columns in Table I list the data sources, APIs, clients and their scales. The columns *snp* and *dep* list the number of snippets and dependencies in the result. The *time* column lists the average time (in second) spent on each client. Most cases finished quickly, and the worst one of 12 seconds was still acceptable as static analysis.

We use two criteria to evaluate the inference result, i.e., *precision* and *recall*. We quantified precision as the percentage of *correct* snippets. The *incorrect* snippets are the ones that lack necessary requisite snippets, auxiliary data,

or instructions. We quantified recall as the rate between current snippets and all the required ones. We calculated precision and recall by manually examining the current snippets and the client source code, with the help of client documents. Due to the big source code scale, for case #3 and #5, the recall values are only approximate, because we cannot guarantee to find out all invocations via manual code reading. The columns `precision` and `recall` list these two criteria. In average, the precise is 89%, with 6 out of 8 cases above 90%, and the recall is 95%, with 7 above 90%.

The incorrect and missing snippets in these cases are summarized as follows. The 2 incorrect snippets of Case #1 are caused by using the instance of a client-defined class as parameter to invoke an API method. Case #3 has the same problem. The incorrect snippets of #2 are caused by wrapping the data into an array before invoking the API. Similar situation also appears in #6, which restore the retrieved data in collections first before using them. For case #4, the client uses the same API invocation to get 6 different types of data elements, and then checks one of their attributes to dispatch the further usage. For such a situation, a necessary instruction exists in the control flow rather than the data flow, and thus we missed this instruction when slicing the code snippets. To fix this fault, we need to split the generic snippet into 6 different ones, and thus we regard it to have 5 missed snippets. We also find about 20 of such generic invocation patterns in case #3. Case #5 has incorrect snippets because it uses the input value to differentiate the type of retrieved data. The missed snippets in case #5 and #7 are caused by the situation that the library classes used by the clients also contain the invocation to the DOM API, and thus the inference omits these invocations.

The experiment results lead to the following conclusions. First, *the current configuration of points-to analysis is proper for the inference of data access*. On one hand, the context combined by single receiver and single call site is enough, because no incorrect or missed snippets in our experiments are caused by fault points-to mappings. On the other hand, the current context is necessary. In an incomplete investigation of the subject clients, we found the code pieces from case 1, 4 and 8 that would cause incorrect snippets if without receiver- or callsite-sensitivity. Regarding the performance, although the current context is already fine-grained, the analysis time is acceptable, thanks to the strategy to regard API and library invocation results as objects and avoid going into the details of APIs and libraries. Second, *data-flow analysis is insufficient for data access inference*. For the clients that differentiate the types of data elements by checking special attribute values (such as Case #4), the control flow should also be included. Third, *ignoring the library code may lead to inference errors, but the probability is small*. In our experiments, it only causes missed snippets in Case #5 and #7. Finally, *static analysis is not enough in certain cases*. In our experiments, the

#	source	API	client	kloc	snp	dep	time	precision	recall
1	Exists	JMX	exist-mgmt(exist.sourceforge.net)	5.6	54	58	1.5	96%	100%
2	JOnAS	JMX	CarteBlanche(code.google.com/p/carteblanchesupervision/)	5.3	34	41	1.3	91%	100%
3	JOnAS	JMX	jonasAdmin(jonas.ow2.org/JOnAS_4_7/doc/Admin.html)	22.5	354	316	12.5	92%	≈95%
4	JBoss	JMX	simplejboss(code.google.com/p/simplejbossstatus/)	5.8	30	50	1.8	70%	87%
5	Flickr	DOM	Flickrj(flickrj.sourceforge.net/)	14.9	313	287	7.5	89%	≈92%
6	MSBib	DOM	JabRef(jabref.sourceforge.net/)	6.8	525	524	2.0	97%	100%
7	GeoRss	DOM	GeoChat(code.google.com/p/geochat/)	11.8	146	142	3.1	100%	91%
8	FriendFeeder	JSON	ff-java-api(code.google.com/p/friendfeed- java-api/)	1.4	128	123	3.3	98%	100%

major reason of incorrect snippets are the lack of execution contexts. e.g., some clients use arrays to store intermediate results, and some depend on external values to decide data types. Such runtime contexts are hard to establish by merely static analysis.

B. Inspection cases

We used the inferred summoning snippets to inspect the above clients, and found the following two problems.

Overclaim of data coverage. The client of #6 is a sub function of JabRef, which imports bibliography data from XML files in MS Office bib format. The developer claimed that it fully supports the MSBib format. However, after comparing the inferred summoning snippets with the official XML schema [11], we found out that the snippets reflect 76 out of all the 77 types of XML elements, without one named `RefOrder`. It could be a good hint if the developer would consider to enrich the import function and achieving really full support for MS Bib format.

Version incompatibility. In the above experiments, there are two clients on JOnAS, i.e., jonasAdmin (#3) and CarteBlanche (#2). The former is a default web management console on JOnAS 4.7. The latter is a deployment description tool. In its comment, the developer declared that CarteBlanche applies to JOnAS 5.1 or later, but did not explain why it is incompatible with older versions. Using the summoning snippets, we found out the data types used by it that is not supported in older versions of JOnAS. We achieved this by comparing its summoning snippets with the ones of jonasAdmin, because as a default management console, jonasAdmin should cover all kinds of data elements in JOnAS 4.7. The comparison showed that CarteBlanche has an extra snippets named `DeploymentPlan`, as well as five other dependent snippets.

During the inspection process, we found the following advantages of summoning snippets. First, they showed all the accessed data types in one graph, and thus liberated us from reading the raw source code. Second, the DAGs were intuitive for the browse of data access. For the case of JabRef, the graph of its snippets had exactly the same outline as the MSBib XML schema, and it was easy for us to start the inspection from the `Entry` snippet that locates in the key position in the graph. Third, as the summoning snippets

were implemented in the form of standard EMF models, we can utilize the existing EMF tools such as OCL to preprocess the result. After locating the key snippet of the JabRef case, we wrote a simple OCL query to get all snippets depending on it, and print their names as sorted. The major deficiency is that the inference cannot ensure completeness, and thus the conclusion from the summoning snippets can be only used as a guidance.

C. User study

We invited volunteers to write programs on the data sources, either with or without the summoning snippets. We chose four data sources, 1, 2, 5 and 6, and for each of them, we designed 2 to 4 problems about retrieving data from their source, to simulate adding new data access logics into the existing clients. Below are four problems for Case 1.

- **P1.** Get the ID of a query under processing.
- **P2.** Get the ID of a running job.
- **P3.** Get the job description.
- **P4.** Get the error code reported by the database.

We invited 6 volunteers, including 3 graduate students, 1 undergraduate student and 2 software engineers. Four of them were familiar with Java, while the other two were not. No volunteer was familiar with the data sources. All the volunteers were asked to resolve all the 12 problems by writing code on the API, and record the time they spent on each problem. Each volunteer had two of the four data sources equipped with inferred summoning snippets. The time spent on each problem was recorded from the moment they read the problem until they finished the programming. If a volunteer spent a period of time to understand the data source or read the snippets before resolving any of its problems, then this time was equally distributed to the problems. For each volunteer, we spent two to five minutes training them to use the summoning snippets, and this time was distributed into all their problems. If a problem was not resolved in 20 minutes, we marked it as unsolved.

Figure 6 shows the time each volunteer (*A-F*) spent on each problem (*P1-P12*). For each problem, there were three volunteers with snippets (shown by yellow circle), and three others without (blue square). Without snippets, most volunteers started from reading the API clients. They searched the code using the keywords they deduced from the

problem descriptions, and then traced the code following cross references. Most volunteers agreed that the main trouble was the big code size. *F* also complained that since she was used to start each problem from searching the code, she did not notice that some problems (such as Q2 and Q3) are tightly related, and wasted time on searching from scratch every time. *D* found the sample XML data for *P11* and *P12*, but being not familiar with DOM, he was blocked by trying to parse the string into an XML document. *E* started *P5* to *P7* from searching on Google, because he never used Java and was afraid to read the client source code. He spent nearly an hour without good progress and marked the problems as unsolved. On the contrary, with summoning snippets, volunteers all started from browsing them. They found the proper snippets by browsing the DAG, and use the composed code slice as a draft to write their code.

From the user study, we have the following empirical findings. First, *the summoning snippets improved the programing efficiency for data retrieval.* The direct reason is that our snippets provided draft code for retrieving each type of data, liberating developers from writing code or extracting examples from the clients. However, since these problems require only small amounts of code (about 10 lines in average per problem), the help from code generation should not be so significant. The other reason was that the intuitive DAG of snippets help developers understand the data access and found the path leading to the target data. Second, *the summoning snippets helped the reuse of code between problems.* For the problems related to previous ones (especially *P3*), the snippets provided most significant improvement. That was because the DAG clearly showed the common ancestors between snippets, and this directly led the developers to reuse the code for the ancestor type when solving the problems. Third, *the inferred summoning snippets helped reduce the dependence on developers' experience.* We noticed that without summoning snippets, the average time of different volunteers differed largely, from 11 to 20 minutes, but with reference snippets, it was only between 7 to 11. The case of *F* is the most interesting. Unfamiliar to Java, she spent the longest time on *P1-P4*. But with the snippets, she even managed to be the best on *P11* and *P12*. One possible reason was that the DAG representation of summoning snippets was independent to techniques and languages.

VIII. RELATED WORK

Our paper is related to the type inference of untyped data sources. Existing approaches usually do this inference from data samples, such as Garofalakis et al's XTRACT [12] on XML documents, Asai et al.'s approach [13] on Web pages, and Fisher et al.'s PADS [1][10] on plain-text data. These approaches focus on "how the data are organized", whereas our work is from the perspective about "how a program access the data", in order to provide direct
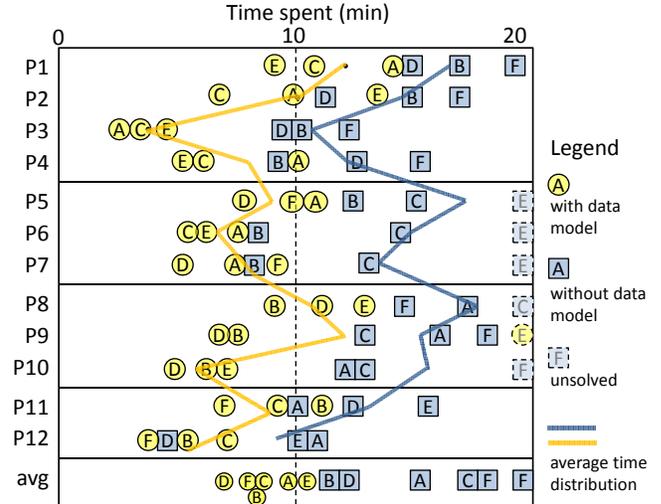

Figure 6.   Time spent by volunteers

assistance for maintaining data processing programs. From this perspective, we take source code but not data samples as the input, and the output is not the complete data schema, but the one covered by the client. Another difference is that we also extract the code slice for each type of data. Jahani et al.'s work on analyzing the data access source code to improve the performance of data retrieval [14] is close to our idea, but their objective of analysis is to identify some specific patterns but not recover the whole data access.

Static code analysis is widely used in inferring the API usages from its clients. Approaches on "mining specifications" [15] analyze the API usage of clients to identify the illegal or dangerous API usage. Antkiewicz et al. [16] analyze the clients on framework APIs, and summarize what API features are used by the clients, and how to use these features. Mandelin et al. [2] and [3] abstract the API usage as the tasks to instantiate specific API classes from existing ones, and provide different techniques to extract such snippets from sample clients of the API. These approaches are all based on the class hierarchies of the APIs. In other words, they assume that each Java class in the API has a unique meaning and usage. This work targets at the generic data processing APIs, where one Java class in the API may represent different kinds of data elements. Therefore, we have to differentiate the usages according to the parameters used by the API invocation, and this requires more sophisticated analysis to clarify the origin of each parameter variable.

Our analysis approach is based on existing static analysis techniques. We utilize the basic points-to analysis theory presented by Grove et al. [5] and its extension of receiver sensitivity presented by Milanova et al. [7]. The idea of regarding invocation results as objects is related to Chatterjee et al.'s relevant context inference [17]. We believe the deeper points-to analysis techniques, such as the con-

sideration of reflections [18], will improve the application scope of our approach, but we leave this for future work. The inference of different API invocations is related to the def-use analysis [6], but we also need to clarify all the individual and sufficient compositions of the used objects. The construction of snippet contents is a typical data-flow-based code slicing [4]. But in this paper, we do slicing based on the result of points-to analysis, rather than constructing the program dependency graph [19], in order to deal with the fields and virtual methods in an object-oriented language. Tan et al. [20] and Willmore et al. [21] extend the program dependency graph with the state of database, in order to extract the database access slices considering the data types. However, since we target the generic APIs, we do not know the effect of different API invocations on the system state, but can only utilize the dependency between retrieved data elements and the subsequent invocations.

This work originates from our initial investigation on management APIs [22], but in the previous paper, we only infer data types mentioned by the API clients, and the goal is to assist the construction of runtime models. In this paper we propose a new way to represent the data access in order to support the maintenance and development of data processing programs. From the technical perspective, in this paper, we applied points-to analysis on the client source code, so that the approach applies to the complex object-oriented programs.

IX. CONCLUSION

This paper reported a new approach to representing and inferring the data access of the clients of generic APIs. We represent all the types of data accessed by a client as summoning snippets, and record the code slice to retrieve each type of data. We provide an automated approach to inferring the summoning snippets, based on points-to analysis and code slicing. Our experiments showed that this approach is able to infer the data access from many open source clients, and the inferred summoning snippets are useful for the maintenance of these clients.

We currently infer the API usages only along data flows, and as a static analysis, we cannot deal with the intermediate results stored in collections or arrays, or the invocations that depends on external values. We will address these issues to improve the accuracy. Currently, we assume different API invocations retrieve different types of data. As a future plan, we will find the empirical rules to identify the code snippets in the client that retrieve the same kind of data from different ways of API invocations.

REFERENCES

[1] K. Fisher and R. Gruber, "PADS: A domain specific language for processing ad hoc data," in *PLDI*, 2005, pp. 295–304.

[2] D. Mandelin, L. Xu, R. Bodik, and D. Kimelman, "Jungloid mining: helping to navigate the api jungle," in *PLDI*, 2005, pp. 48–61.

[3] N. Sahavechaphan and K. Claypool, "XSnippet: Mining for sample code," in *OOPSLA*, 2006, pp. 413–430.

[4] M. Weiser, "Program slicing," in *ICSE*, 1981, pp. 439–449.

[5] D. Grove and C. Chambers, "A framework for call graph construction algorithms," *TOPLAS*, vol. 23, no. 6, pp. 685–746, 2001.

[6] H. Pande, W. Landi, and B. Ryder, "Interprocedural def-use associations for c systems with single level pointers," *TSE*, vol. 20, no. 5, pp. 385–403, 2002.

[7] A. Milanova, A. Rountev, and B. Ryder, "Parameterized object sensitivity for points-to analysis for java," *TOSEM*, vol. 14, no. 1, pp. 1–41, 2005.

[8] Watson, T.J.: Libraries for Analysis (WALA), http://wala.sf.net.

[9] F. Budinsky, S. Brodsky, and E. Merks, *Eclipse Modeling Framework*, project address: http://www.eclipse.org/modeling/emf.

[10] K. Fisher, D. Walker, K. Zhu, and P. White, "From dirt to shovels: Fully automatic tool generation from ad hoc data," in *POPL*, 2008, pp. 421–434.

[11] MSBib XML Schema, http://www.schemacentral.com/sc/ooxml/s-shared-bibliography.xsd.html.

[12] M. Garofalakis, A. Gionis, R. Rastogi, S. Seshadri, and K. Shim, "XTRACT: a system for extracting document type descriptors from xml documents," *ACM SIGMOD Record*, vol. 29, no. 2, pp. 165–176, 2000.

[13] T. Asai, K. Abe, S. Kawasoe, H. Arimura, H. Sakamoto, and S. Arikawa, "Efficient substructure discovery from large semi-structured data," in *ICDM*, 2002, pp. 158–174.

[14] E. Jahani, M. Cafarella, and C. Ré, "Automatic optimization for mapreduce programs," *Proceedings of the VLDB Endowment*, vol. 4, no. 6, pp. 385–396, 2011.

[15] M. Acharya, T. Xie, J. Pei, and J. Xu, "Mining API patterns as partial orders from source code: from usage scenarios to specifications."

[16] M. Antkiewicz, K. Czarnecki, and M. Stephan, "Engineering of framework-specific modeling languages," *TSE*, vol. 35, no. 6, pp. 795–824, 2009.

[17] R. Chatterjee, B. Ryder, and W. Landi, "Relevant context inference," in *POPL*, 1999, pp. 133–146.

[18] V. B. Livshits, A. V. Nori, S. K. Rajamani, and A. Banerjee, "Merlin: specification inference for explicit information flow problems," in *PLDI*, 2009, pp. 75–86.

[19] S. Horwitz and T. Reps, "The use of program dependence graphs in software engineering," in *ICSE*, 1992, pp. 392–411.

[20] H. Tan, T. Ling, and C. Goh, "Exploring into programs for the recovery of data dependencies designed," *TKDE*, vol. 14, no. 4, pp. 825–835, 2002.

[21] D. Willmor, S. Embury, and J. Shao, "Program slicing in the presence of database state," in *ICSM*. IEEE, 2004, pp. 448–452.

[22] H. Song, G. Huang, Y. Xiong, F. Chauvel, Y. Sun, and H. Mei, "Inferring meta-models for runtime system data from the clients of management APIs," *MoDELS*, pp. 168–182, 2010.