
Software Engineering Group
Department of Computer Science
Nanjing University
http://seg.nju.edu.cn

**Technical Report No. NJU-SEG-2015-IJ-002**

**2015-IJ-002**

# A declarative approach for Java code instrumentation

Tian Zhang, XiaomeiZheng, Yan Zhang, Jianhua Zhao, Xuandong Li

Software Quality Journal 2015

Software Qual J (2015) 23:143–170
DOI 10.1007/s11219-013-9220-4

# A declarative approach for Java code instrumentation

**Tian Zhang · Xiaomei Zheng · Yan Zhang · Jianhua Zhao · Xuandong Li**

Published online: 11 September 2013


**Abstract** Source instrumentation plays an important role in dynamic program analysis. However, current instrumentation implementations require programmers to write ad hoc rules that are often too complex to use and maintain. To address this complexity, we divide the task of source instrumentation into two steps: first, the source points are queried, into which code fragments should be planted; secondly, the code fragments including contextual information are generated and planted into source code through the queried points. According to this idea, we present a new method based on declarative code queries, which makes it easier to specify instrumentations using contextual information collected from expressive code queries. The JIns language provided by our method is constructed following an SQL-like style, which is well known and widely used by programmers. We evaluate the method in terms of the reduced complexity of instrumentation specifications for several common instrumentation tasks.

**Keywords** Source query · Code instrumentation · Java

## 1 Introduction

Most dynamic program analysis requires *instrumentation*, that is, inserting executable code fragments into the investigated program to perceive its behavior at runtime. Such

T. Zhang (✉) · J. Zhao · X. Li
National Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China
e-mail: ztluck@nju.edu.cn

X. Zheng
School of Information Technology, Nanjing University of Chinese Medicine, Nanjing, China
e-mail: zhengxiaomei@njutcm.edu.cn

Y. Zhang
Department of Computer Science and Technology, Beijing Electronic Science and Technology Institute, Beijing, China
e-mail: zhangyan@sei.pku.edu.cn

instrumentation techniques have been widely used in various software quality assurance tasks such as debugging, testing, code review, and end-user behavior analysis.

Instrumentation can be accomplished either by modifying the binary code or by modifying the source code. Although mostly used in practice, binary code instrumentation requires programmers to consider low-level details of the implementation, such as method and data memory addresses, and their relocating offsets before or after the instrumentation. Therefore, high-level programming language source code instrumentation is preferred by programmers given that it can be automated to avoid introducing syntax, type checking, and semantic errors.

However, current source code instrumentation tools are often too complex to specify an instrumentation. Specifically, we consider two related requirements for source code instrumentations: (1) programmers shall be able to flexibly select the points of program execution to anchor the instrumented code; (2) an instrument statement shall be sensitive to the context of selected anchors. Accordingly, our instrumentation solution has two parts:

1. Locating or anchoring an execution point for instrumentation; and
2. Constructing an instrumentation code fragment using the contexts of the selected points.

In this paper, we proposed a declarative approach to specify any source code instrumentation using a language, JIns, which enables programmers both select instrumentation locations and construct the context-sensitive instrumentation code fragments.

Similar to what the structured query language (SQL) does for database queries, the JIns language simplifies the task of specifying the search requirements for code structures and defines a view of the code. The query-based source instrumentation helps one anchor the source location for the instrumentation, as well as collect context-dependent information from the code. When the search returns, instrumentation statements can be composed including the searching results.

Comparing to the state of the art of source instrumentation techniques, the JIns approach has two major advantages:

- *Code queries are declarative and expressive*. Using predicate logic expressions, JIns provides expressive yet simple query rules for code, making it suitable to specify the search requirements declaratively that can match with statements.
- *Instrumentations are fine-grained and context-sensitive*. The JIns language expresses instrumentation as parameterized templates whose arguments are replaced by the matching query results. This allows more flexible and fine-grain (statement-level) instrumentation anchors to be located, and the results of the search are used to construct some context-sensitive instrumentations.

The proposed declarative query-based instrumentation method has been implemented for the Java programming language, and the corresponding prototype instrumentation tool is provided as a set of Eclipse plugins.

The rest of this paper is organized as follows: Sects. 2 and 3 introduce respectively the features of query and instrumentation parts of the JIns language. Section 4 provides a solution to calculate the instrumentation running time. Section 5 details the implementation of JIns language and shows the usage of JIns prototype tool. We evaluate the effectiveness of the JIns approach through two case studies in Sect. 6. In addition, Sect. 7 compares the JIns approach with related work, and finally, Sect. 8 draws conclusion that the declarative JIns approach reduces the complexity of Java code instrumentation, and plans the future work.

**Fig. 1** Properties and relationships in program

```
1 public void show() {
2   for (int i = 0; i < a.size();i++) {
3     a.get(i).trim();
4   }
5 }
```

## 2 The JIns approach: code query

Code query is fundamental to many tasks in software engineering (Verbaere et al. 2008). Most query tools are designed to help developers understanding varied and complex relationships in code structures. One representative tool is grep, a full-text search utility program that searches files or standard inputs for the lines matching some given regular expressions. Although efficient, string-based regular expressions cannot describe certain structural relationships in programs, such as multilevel class inheritances and multilevel method invocations.

### 2.1 Properties and relationships in object-oriented programs

An object-oriented program can be divided into several elements. For example, a Java program consists of classes, interfaces, methods, statements, and objects. Each element has its own properties and several relationships to other elements.

Figure 1 lists a fragment of Java source code. Suppose the statement in Lines 2–4 is identified by *s*, we can use the follow property to indicate that the statement is a for-loop:

```
s.specifyType='for'
```

In the condition part of this for-loop statement, there is an object called a, and suppose the object is identified by *o*. We refer the relationship between the for-loop statement and the object as *use*:

```
(where s.specifyType='for' and o.name='a')
```

### 2.2 The general JIns query language

For arbitrary properties and structures, we need a more expressive query language that can specify any structural relationships in programs. The JIns code query is designed to locate arbitrary instrumentation points of source code in the declarative style similar to that of the SQL.

The language allows programmers to specify the type of target code element to be found such as a class, a method, a statement, or an object. The language also provides programmers a way to express the query conditions that the target code elements should satisfy. To describe complex contexts, JIns provides other code elements in two types of clauses, respectively, for property conditions and relationship conditions. Programmers can combine the conditions with operators, such as logical connectors and quantifiers, such as "&&," "||," "exist," "all".

The following production rules describe the grammar of JIns language:

$$
\begin{aligned}
S &\Rightarrow find\ Id : T\ satisfying\ CS \\
T &\Rightarrow object \\
&\quad | \ statement \\
&\quad | \ method \\
&\quad | \ class \\
&\quad | \ interface \\
CS &\Rightarrow \{exist\ Id : T\}\ \{all\ Id : T\}\ where\ CE \\
CE &\Rightarrow CE\ \&\&\ CE \\
&\quad | \ CE \ || \ CE \\
&\quad | \ !\ CE \\
&\quad | \ (\ CE\ ) \\
&\quad | \ Id.ATT\ =\ 'value' \\
&\quad | \ Id\ \ REL\ \ Id \\
ATT &\Rightarrow \ name \\
&\quad | \ dataType \\
&\quad | \ specificType \\
&\quad | \ returnType \\
&\quad | \ paramsType \\
REL &\Rightarrow extend \\
&\quad | \ use \\
&\quad | \ change \\
&\quad | \ isIn \\
&\quad | \ call
\end{aligned}
$$

A query command has two parts: target declaration and conditions. The query starts with the keyword find, and then, the target code element *Id* of the type *T* is specified in the form of *Id* : *T*. The conditions are defined by *CS* after the keyword satisfying.

The type *T* can be one of object, statement, method, class, or interface. The identifier *Id* is used in the conditions defined in the nonterminal *CE*. All the identifiers in conditions should be declared before being used and unique in the a whole command. For example, "s1:statement" defines a statement element that can be either a single statement or a compound statement, and this identifier "s1" can be used in the following conditions.

The nonterminal *CS* describes search conditions which the target code element should satisfy. The rules are defined based on first-order logic with the quantifiers exist and all, and the expression *CE* holds with code element $Id_i$ whose type is $T_i$.

In the nonterminal *CE*, *Id.Att* =′ *value*′ and $Id_1$ *Rel* $Id_2$ are atomic conditional expressions, which can be used to construct more complex conditions using the operators "!," "||," and "&&," The first atomic expression, *Id.Att* = "value," is a property expression that returns *true* only if the attribute *Att* of the code element *Id* evaluates as "*value*." And the second, $Id_1$ *Rel* $Id_2$, describes the relationship between the code elements $Id_1$ and $Id_2$, which returns *true* only if the relationship *Rel* holds.

### 2.3 JIns language details relating to object-oriented language

In Sect. 2.1, the main elements of object-oriented languages, typically Java, are classified into the five kinds, such as *class*, *interface*, *object*, *statement,* and *method*. We also defined the properties of each element and the corresponding meanings of properties' values.

Object elements have three properties: (1) *name*, representing the identifier of an object in a program; (2) *dataType*, representing the type of an object, for example, if an object is declared as private Map m = new HashMap(), the *dataType* of object m is Map; (3) *specificType*, representing the scope of an object. The value "field" of specificType means

that the object is a field of a class, the value "local" means that the object is a local variable of the program, and the value "parameter" means that the object is a parameter of a method.

The following expression constrains objects with the name "initList,", and at the same time, they are not the parameters being passed in:

```
o.name='initList'
&& !o.specificType='parameter'
```

Statement elements have only one property, *specifyType*, representing the type of a statement in programs. Its value can be "single," meaning that the statement does not contain any substatements, or "while," meaning that the statement is a while loop. Similarly, other values can be used to indicate "'for," "if" statements, etc.

Method elements have four properties: (1) *name*, representing the identifier of a method in programs; (2) *returnType*, representing the type of returned values, e.g., the *returnType* of String.split(String) is "String[]"; (3) *argumentsTypes*, representing the list of arguments' type of a method, for example, "String, String" is the value of *argumentsTypes* of String.replace(String, String); (4) *specificType*, representing the modifier of a method. The above four properties are to be used to describe the signature of a method. For example, int Integer.parseInt(String) can be matched by the following expression:

```
m.name='parseInt'
&& m.returnType='int'
&& m.argumentsType='String'
&& m.specifyType='static␣public'
```

Class and interface elements both have only one property: *name*, representing the identifier of the element.

We also define a series of relationship between different code elements:

- *extends* relationship has three forms: (1) class element *extends* class element, meaning that a class inherits from another; (2) interface element *extends* interface element, meaning that an interface extends another interface; (3) class element *extends* interface element, meaning that a class implements an interface.
- *isIn* relationship has four forms: (1) statement element *isIn* statement element, representing that a statement is in another statement. The first statement can be a single statement or a compound statement, and the second statement must be a compound statement that means the *specifyType* value of the second element cannot be "single"; (2) statement element *isIn* method element, representing that a statement is in a method; (3) method element, statement element or object element *isIn* class element, representing that one of these elements is in a class. For object element, either local object or field can be used in this relationship; and (4) method element *isIn* interface element, representing that a method element is in an interface element.
- *use* relationship has three forms: (1) statement element *use* object element, representing that a statement element use an object element. This relationship is similar to an object *isIn* a statement, but the former does not contain any conditions that an object declared in a declaration statement; (2) method element *use* object element, representing that a method element uses an object element. This relationship often used to describe some

methods using some constants defined in other class; (3) class element *use* object element, representing that a class element use an object element.

- *change* relationship also has three forms: (1) statement element *change* object element, representing that a statement changes an object's value; (2) method element *change* object element, representing that a method modifies an object's value; and (3) class element *change* object element, representing that a class changes an object's value.
- *call* relationship has two forms: (1) statement element *call* method element, representing that a method element is called in a statement element; (2) method element *call* method element, representing that a method calls another method to accomplish its function.

All these relationships could also be obtained using the static analysis on source code.

## 3 The JIns approach: contextualization

In addition to locating instrumentation points, another difficulty of automated instrumentation is to construct different instrumentation fragments according to the context of their locations in the target programs.

### 3.1 Context-sensitive instrumentation in a motivated example

Suppose that a programmer wants to check whether the for-statements, using objects of List in their loop conditions, change the size of these lists after looping. Normally, she has to pick up the proper for-statements, i.e., those using objects whose type is List in the loop conditions. Then, she would like to insert different instrumentation fragments before and after different for-statements. The inserted fragments may vary according to the context of for-statements.

Figure 2 illustrates a sample Java code fragment with two for-statements. According to the requirements, both for-statements are qualified because they use variables a and b, objects of List<String>, in their loop conditions. Hence, the instrumentation code fragments are to be inserted before and after each of them, respectively.

It is typical to check the equality of two variables using the assertion technique. Therefore, some intermediate variables and assertions need to be generated and inserted to the corresponding instrumentation points. The first for-statement uses java.util.List objects named a and b (Fig. 2, Line 6) in its loop condition, and the second uses the object named b (Fig. 2, Line 13). The final code fragment with inserted instrumentations is shown in Fig. 3.

### 3.2 Contextualizing the instrumentation templates

This section introduces a method for constructing instrumentation statements using the set of execution points obtained by code queries in Sect. 2.

Since an instrumentation statement must be relevant to the program context specified by a query, we extend the query part of JIns language to integrate a template of instrumentation statements. The extension takes advantage of the queried results to substitute placeholders in the template with code element variables in the query.

The example shown in Fig. 3 illustrates the instrumented code with different insertion fragments for the different for-statements, where Lines 7–8 and Lines 14–17 are the first

**Fig. 2** For-statements that use List objects

```
1 private List<String> a, b;
2 //...
3 public void fun() {
4   //...
5   for (int i = 0, j = 0
6     ; i < a.size() && j < b.size()
7     ; ++i, ++j) {
8     String tmp1 = a.get(i).trim();
9     String tmp2 = b.get(j).trim();
10    b.set(j, tmp1 + tmp2);
11  }
12  //...
13  for (int i = 0; i < b.size(); ++i) {
14    System.out.println(b.get(i));
15  }
16 }
```

**Fig. 3** For-statements with instrumentation

```
1 public void fun() {
2   int __lena1__;
3   int __lenb1__;
4   int __lena2__;
5   int __lenb2__;
6   //...
7   __lena1__ = a.size();
8   __lenb1__ = b.size();
9   for (int i = 0, j = 0
10    ; i < a.size() && j < b.size()
11    ; ++i, ++j) {
12    // the same as Lines 8-10 in Fig. 2
13  }
14  __lena2__ = a.size();
15  __lenb2__ = b.size();
16  assert __lena1__==__lena2__;
17  assert __lenb1__==__lenb2__;
18  // same as Line 12 in Fig. 2
19  __lenb1__ = b.size();
20  for (int i = 0; i < b.size(); ++i) {
21    // the same as Line 14 in Fig. 2
22  }
23  __lenb2__ = b.size;
24  assert __lenb1__ == __lenb2__;
25 }
```

part of instrumentation, and Line 19 and Lines 23–24 are the second part of the instrumentation. The differences between the two parts of instrumentations are mainly concerning to the identifiers of the java.util.List objects inside the for-statements. The reason is obvious: the contexts of instrumentation points are different. Therefore, the context information is required to generate the corresponding instrumentation code.

In this case, the java.util.List objects, a and b, could be captured by the JIns query command shown in Fig. 4. The quantified variable, declared by exist o:object, will be bound to the objects matched by the conditions defined in where block. Line 2–5 defines four conditions with the variable o for java.util.List objects in the for-statements.

**Fig. 4** Query command example

```
1 find s:statement satisfying exist o:object
2   where o.dataType="java.lang.List"
3     && o.specificType="field"
4     && s.specificType="for"
5     && s use o
```

As a result, the first for-statement matches with the objects a and b, and the second for-statement matches with the object b. In this way, the contextualization requirement of source instrumentation is solved by recording the context information.

### 3.3 Specifying templates with instrumentation language

We make the following extension to the production rules in Sect. 2.2:

$$
\begin{aligned}
S &\Rightarrow find\ Id : T\ satisfying\ CS\ IS \\
IS &\Rightarrow insert\ before\ Id\ IN\ IS \\
&\quad | \ insert\ after\ Id\ IN\ IS \\
&\quad | \ NULL \\
IN &\Rightarrow 'STR' \\
STR &\Rightarrow \ String\ STR \\
&\quad | \ \%Id.Att\%STR \\
&\quad | \ String \\
&\quad | \ \%Id.Att\%
\end{aligned}
$$

The nonterminal *IS* represents an instrumentation command. It starts with the keyword insert. The keywords before and after indicate that an instrumentation template will be inserted before and after the code element *Id*, defined earlier in the query part of JIns command. The nonterminal *IN* represents an instrumentation statement template, which is between a pair of single quotation marks. The content between the two percent signs will be substituted by the value of code element attributes.

The instrumentation template provides a way for constructing the instrumentation statements. It uses the code elements collected in query process and substitutes the arguments in templates with the values of those matched elements. The query language should have defined all these elements in the form like *Id*:*T*, while the arguments in templates must be declared in the form of *Id.Att*.

```
1 find s:statement
2 satisfying exist o:object
3 where o.dataType='java.util.List'
4   && o.specificType='field'
5   && s.specificType='for'
6   && s use o
7 insert before s
8   'int __len%o.name%1__=%o.name%.size();'
9 insert after s
10  'int __len%o.name%2__=%o.name%.size();
11  assert(__len%o.name%1__==__len%o.name%2__);'
```

**Fig. 5** Instrumentation command example

The example in Fig. 5 shows a command that can achieve the required instrumentation in Fig. 3. The instrumentation templates with parameters containing the context information could be used to verify whether the size of java.util.List objects have been changed after the loop:

```
int __len%o.name%1__=%o.name%.size();
int __len%o.name%2__=%o.name%.size();
assert(__len%o.name%1__==__len%o.name%2__);
```

In Fig. 5, Lines 1–6 are command lines for matching the source code and locating the instrumentation points. It is the query part of the whole command. In this part, some instrumentation variables are declared, such as s:statement, o:object, which is to be used to generate instrumentation code. Following the code, query part is the instrumentation part starting with the keyword insert, Line 7–11. The location anchor for the code to be inserted is the variable s. When generating insertion code, Line 8 and 10–11, o.name will be replaced by the real objects bound in the matching process of the query command.

## 4 Time measurement of instrumentation

In many domains, specially real-time and embedded systems, time performance of implanted fragments needs to be paid much attention during testing or analyzing phases. For example, when a real-time programmer wants to extract the run-time path of an embedded system, she may plant some "*print-out*" code fragments in the source and then execute it. Normally, the traces are obtained so that the run-time information could be analyzed. Unfortunately, the planted code also costs time to execute, which may disturb the original execution and even cause some unexpected results.

More interesting is that the time effect may vary largely according to the location where to be planted. These locations could be indicated as *time sensitive points*. Tang et al. (2000) gave the detailed discussion on how to detect such kind of time sensitive points by instrumenting fragments on source code.

Therefore, it makes sense if the execution time of every single planted fragment could be measured when dynamically executing the instrumented programs, especially in the real-time-related domains. Though there are still some other aspects, such as memory occupancy and potential bad effects to data integrity, we focus on time dimension and provide in JIns the way to measure the running time of its constructed fragments. This section provides a method to calculate the running time of instrumentation, so as to evaluate their time performance.

### 4.1 Calculating the running time of instrumentation

Currently, execution time is usually calculated using static analysis techniques. For example, static methods and tools have dominated WCET(Worst-Case Execution Time) estimation of software in hard real-time systems (Wilhelm et al. 2008). It is believed that the best way to acquire the accurate executing time of a program is calculating at hardware level, i.e., calculating the instructions on specific CPUs.

Despite many of their advantages, static analysis techniques usually offer heavyweight methods to calculate and estimate the running time of instrumented code fragments. In

addition, the result of calculation is still the estimation, and real executions may vary in different scenarios. Moreover, when testing and debugging programs, the actual and instant running time that the planted fragments cost is more meaningful to developers. Therefore, we provide a lightweight but effective mechanism in JIns to calculate the running time of instrumentations.

The command calculating time consumption is provided by extending JIns with a keyword calculate. Users can add calculate after the insert clause to acquire the running time of inserted fragment. As shown in Fig. 6, on Line 9 the calculate keyword is used to specify that the planted code of Line 8 will be calculated the time of its execution.

To achieve this aim, we design a timer module that records running time and calculates the execution time of flagged fragments. The timer module is implemented by a RunTime class containing mainly two public methods, start() and end(). Method start() is used to record the time stamp before instrumentation, and end() will calculate the time of instrumentation cost and record it. We implement both methods with System.nanoTime() of JDK API, which returns a long integer representing the current time of system in nanosecond. When the JIns interpreter meets the keyword calculate, it will insert RunTime.start() before and RunTime.end() after the inserted code.

```
1 find s:statement
2 satisfying exist o:object
3 where o.dataType='java.util.List'
4   && o.specificType='field'
5   && s.specificType='for'
6   && s use o
7 insert before s
8   'int __len%o.name%1__=%o.name%.size();'
9   calculate
10 insert after s
11   'int __len%o.name%2__=%o.name%.size();
12   assert(__len%o.name%1__==__len%o.name%2__);'
```

**Fig. 6** JIns command with time calculation (Method 1)

```
1 find s:statement
2 satisfying exist o:object
3 where o.dataType='java.util.List'
4   && o.specificType='field'
5   && s.specificType='for'
6   && s use o
7 insert before s
8   'RunTime.start("Before %s.content%");
9   int __len%o.name%1__=%o.name%.size();
10   RunTime.end("Before %s.content%");'
11 insert after s
12   'RunTime.start("After %s.content%");
13   int __len%o.name%2__=%o.name%.size();
14   assert(__len%o.name%1__==__len%o.name%2__);
15   RunTime.end("After %s.content%");'
```

**Fig. 7** JIns command with time calculation (Method 2)

```
1 find s:statement
2 satisfying exist o:object
3 where o.dataType='java.util.List'
4   && o.specificType='field'
5   && s.specificType='for'
6   && s use o
7 insert before s
8   'int __len%o.name%1__=%o.name%.size();'
9   calculate
10 insert after s
11   'int __len%o.name%2__=%o.name%.size();
12   assert(__len%o.name%1__==__len%o.name%2__);'
13   calculate
```

**Fig. 8** JIns command with calculation

```
1 public void fun() {
2   RunTime.start("Before for");
3   int __lena1__ = a.size();
4   RunTime.end("Before for");
5   for (int i = 0, j = 0; i < a.size();
6     ++i, ++j) {
7     String tmp1 = ((String)a.get(i)).trim();
8     String tmp2 = ((String)b.get(j)).trim();
9     b.set(j, tmp1 + tmp2);
10  }
11  RunTime.start("After for");
12  int __lena2__ = a.size();
13  assert __lena1__ == __lena2__;
14  RunTime.end("After for");
15  RunTime.start("Before for");
16  int __lenb1__ = b.size();
17  RunTime.end("Before for");
18  for (int i = 0; i < b.size(); ++i) {
19    System.out.println(b.get(i));
20  }
21  RunTime.start("After for");
22  int __lenb2__ = b.size();
23  assert __lenb1__ == __lenb2__;
24  RunTime.end("After for");
25 }
```

**Fig. 9** Target code with instrumentation

Users could also declare explicitly RunTime.start() and RunTime.end() in a JIns command to trace the exact time instance and then calculate the corresponding running time. As illustrated in Fig. 7, Line 8–10 and Line 12–15 could record and calculate the executing time of Line 9 and Line 13–14, respectively.

**Table 1** Running time of instrumentations in Fig. 9

| ID | Run 1st (ns) | Run 2nd (ns) | Run 3th (ns) | Run 4th (ns) | Run 5th (ns) | Avg. time (ns) | Proportion (%) |
|---|---|---|---|---|---|---|---|
| No. 1 | 41,489 | 40,207 | 74,424 | 41,061 | 40,634 | 47,563 | 16 |
| No. 2 | 7,699 | 8,555 | 8,126 | 8,127 | 8,126 | 8,127 | 3 |
| No. 3 | 17,536 | 17,536 | 18,820 | 17,536 | 17,536 | 17,793 | 6 |
| No. 4 | 7,271 | 7,271 | 7,272 | 7,271 | 7,271 | 7,271 | 2 |
| Total | 287,431 | 306,679 | 319,511 | 292,136 | 299,408 | 301,033 | |

### 4.2 A sample of time calculation

In this section, a simple sample of JIns instrumentation command is presented containing time calculation clause for the planting code. In addition, the statistics of the running time are also demonstrated. In fact, the fragments of this sample have been presented in some of the above sections, and hence, we do not explain the command lines and focus on the time calculation. The whole JIns command is shown in Fig. 8. The target source to be instrumented by this JIns command is the example in Fig. 2.

Figure 9 shows the instrumented code after the execution of the command in Fig. 8. When executing the program, the running time of each instrumented fragment is calculated. Based on every single time duration of planted fragments, the statistics of the entire instrumentation with proportions could be given, as illustrated in Table 1.

Table 1 lists the running time of the four instrumentations in Fig. 9 at Line 2–4 (denoted as No.1), Line 11–14 (No. 2), Line 14–15 (No. 3), and Line 21–24 (No. 4), respectively. From Table 1, it could be easily found that the first instrumentation code costs the most time, on the average 16 % of the total executing time. This result is helpful for developers to know clearly on what degree each instrumented code disturbs the original source along the time dimension. In this case, the location of the first instrumented code is also known as the *time sensitive points* (Tang et al. 2000). This information is also very useful in maintaining the system in future.

## 5 JIns implementation

The JIns prototype based on Java programming language is composed of four components: *Source Information Generation*, *Code Query*, *Instrumentation Statement Customization*, and *Instrumentation*. The *Source Information Generation* component collects information from Java source code and builds the intermediate structures of code. After analyzing code query commands, the *Code Query* component processes the query and obtains the query results with a set of contextual code elements. Then, the *Instrumentation Statement Customization* component generates concrete statements from the specified JIns template that programmers defined. Finally, the *Instrumentation* inserts concrete statements into the target source code.

### 5.1 Source information generation

We designed an element collector EC to extract and store all information in program, such as objects, statements, methods, classes, and interfaces from the abstract syntax trees

(ASTs) of Java source code. This information also includes the location of every code element such as offset of the element in source file, which is to be used in the later navigation. The data structure *CodeInfo* is similar to an AST, except that some unnecessary information is omitted. Inside the *CodeInfo*, we designed five indices in order to speed up searching process.

Fig. 10 shows the process of *Source Information Generation*. Initially, it generates all ASTs from source files. Every node in an AST stands for a code structure. If a node is one of the five structures, i.e., *object*, *statement*, *method*, *class*, and *interface*, the corresponding data structure will be created in *EC*. Then, the sub-AST will be traversed to find other code elements that have relationships with the current node.

**Algorithm 1** Parsing and preprocessing

**Input:**
Query Command, $S$
**Output:**
An instance of Query and named $Q$

```
1: Match the keyword find
2: Match the target element, and store it into Q.target
3: Match the keyword satisfying
4: while Match the keyword exist ≠ null || Match the keyword all ≠ null do
5:     Construct the predicate object from the string after the keyword
6:     Store the predicate into the list which named Q.predicates
7: end while
8: Match the keyword where
9: while !finish the scanning process do
10:    if Match ATT conditional expression then
11:        Construct an instance of AttrCondition
12:        Store it into the list which named Q.conditionLex
13:    else if Match REL conditional expression then
14:        Construct an instance of RelCondition
15:        Store it into the list which named Q.condidionLex
16:    else if Match operator then
17:        Construct an instance of Operator
18:        Store it into the list which named Q.condidionLex
19:    else
20:        return
21:    end if
22: end while
23: return Q
```

*Parsing and Preprocessing* The process of parsing and preprocessing scans the query command from the head of string and check if it is in coincidence with the grammar. Algorithm 1 shows the main steps of the process. At the same time, all the declarations of elements, in the form of "*Id:T,*" are recorded during the process. More specifically, the alias name *Id* and type *T* are recorded and capsulated into the separated elements so as to be searched respectively. After all, the process transforms the conditional expression *CE* to a list that contains *Rel*, *Attr,* and operators.

A Java class Query is designed to encapsulate the searching functions of a JIns command, which is the implementation of the JIns query command described in Sect. 2. In the class Query, Query.target represents the code elements which should be found, Query.predicate stores all predicates used in the conditional expressions, and Query.conditionLex stores all the conditional expressions.

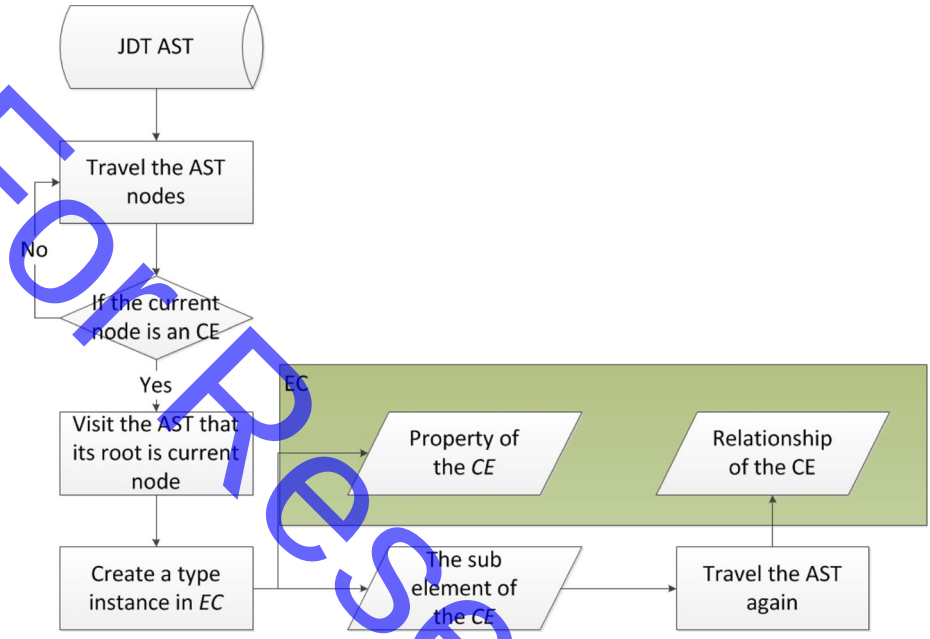



**Fig. 10** Process of source information generation

*Querying the Target Code Element* If *Grammar Check and Preprocess* are finished without finding any syntax errors, querying process will then be started. This process follows the conditions that are specified in the query command *S* to search the target element set. In the previous step, *EC* has collected five types of code elements, and each of the same type is in one of the five index lists. The query process constructs a set and follows the priority of operators to compute the intersection of sets and the union of sets, etc.

A list of conditional expressions is also built in the previous step. So we use *Operator Precedence Grammar* to analyze the list and conduct relevant set operation. First, we define priority of every operator as shown in Table 2. Then, Algorithm 2 is used to analyze the list.

After parsing source code, the tool builds a list of sets of code elements, such as $S_0$, $S_1 \ldots S_n$. Each set is a copy of *EC* that contains some information we needed. Then, it computes every condition according to the priority of operators. For example, suppose that the target element (to be matched in the source code) is defined as "clazz:class," one of the context variables is defined as "obj:object," and the current operation is "clazz.name='Main' && obj.type='int'". Thus, the tool will remove any elements that are not named "Main" in the set $S_{\text{clazz}}$, and remove any elements whose type are not "int" in the set $S_{\text{obj}}$. After that, it will compute the intersection of $S_{\text{clazz}}$ and $S_{\text{obj}}$ to create a new set $S_p$. Similarly, it may compute another conditional expression using the operator "||," e.g., "$S_p \,||\, Q$". After getting the set of $S_q$, it computes the union of $S_p$ and $S_q$. Following the above steps, the final result is obtained.

## 5.2 Instrumentation statement customization

Next, real statements to be inserted into the target source code are constructed according to the instrumentation templates specified by users. First, the set $S_r$ is analyzed which contains all code elements satisfying programmers' query conditions. In the specified templates

**Table 2** Priority of operators

| | ! | \|\| | && | ( | ) | # |
|---|---|---|---|---|---|---|
| ! | < | > | > | < | > | > |
| \|\| | < | > | < | < | > | > |
| && | < | > | < | < | > | > |
| ( | < | < | < | < | = | *E* |
| ) | < | > | > | *E* | > | > |
| # | *E* | *E* | *E* | *E* | *E* | = |

[1] *E* = *Error Status*

defined in user's Jlns command, arguments like "%o.name%" are substituted using values stored in the set $S_r$. The substituted string is the statement to be inserted into the target source code.

**Algorithm 2** Query method based on *Operator Precedence Grammar*

**Input:**
Element collector *EC*
conditional expression list *conditionLex*
**Output:**
The elements which satisfy the users' specification

```
1: Initialize the stack of operator op_stack
2: Initialize the stack of operand r_stack
3: for all Lex object l in the conditionLex do
4:    if l is an ATT or a REL then
5:       Push l into the r_stack
6:    else if l is an Operator then
7:       if l's priority larger than top(op_stack) then
8:          Push l into op_stack
9:       else if l's priority equals with top(op_stack) then
10:         Pop op_stack
11:      else
12:         Pop r_stack, and do set operation
13:      end if
14:   end if
15: end for
16: return element pop up from r_stack
```

## 5.3 Code insertion

In fact, code insertion to the target program is implemented not on the original source code. It is happened on a copy of the target program file. The original program file, assuming the name as"filename.java," is renamed as "filename.java.bak_0" before instrumentations.

After all the customized instrumentation statements being generated, the target source code elements are marked with these instrumentations. These markers record the offsets of statements to be instrumented in the source file as well as the corresponding line numbers of statements.

Programmers are allowed to preview and confirm the inserted contents. If the instrumentation code is correct, they proceed with the real instrumentation. Otherwise, the instrumented code can roll back, and programmers could rewrite the command again.

Users can perform instrumentations many times on a program. Each time the currently file is renamed by adding a suffix "*.bak_i*". Therein, the number "*i*" is the number of times being instrumented. The original program file is renamed by adding the suffix "*.back_0.*" The benefits of the renaming mechanism are as follows: (1) the history of instrumentations could be tracked so as to roll back to a certain version; and (2) in the case of system crash, users could pick up manually the proper version to resume.

### 5.4 Running of the prototype

The JIns prototype tool is currently designed for Java source code instrumentation and also implemented using Java. The tool implementation is based on the extension mechanism of the Eclipse IDE. To analyze Java source code, we reuse the JDT framework and extend its markers mechanism to successfully display instrumentation points.

The task of instrumentation with the prototype begins from an input UI (see Fig. 11a), on which users could enter the JIns query and instrumentation commands in the text box. It also provides a function to specify the scope of the query and instrument based on either all the projects, current project, or the selected resources.

After closing the command dialog, the tool will search all locations satisfying the query command in a table, as shown in Fig. 11b. Each line contains the location Id and type of the relevant code element, content of code element and the concrete information of location. If programmers find that the matched locations are not wanted, they could use trash operation at the top right of viewer to roll back the query operation.

In the editor view, JIns tool shows the planting markers with the source file on the left margin, as illustrated in Fig. 11c. These markers show inserting locations as well as contents to be inserted into the source code. In the editor window containing the source file, these markers are on the left side: red arrows markers indicate that the instrumentation statements will be inserted before the line, while the blue arrow markers indicate that the instrumentation statements will be inserted after the line.

If programmers believe that the locations and the contents of the instrumentations are correct, they could choose the insert operation indicated by the pen icon on top right of the viewer. As a result, code fragments will be inserted into the source file. Figure 11d shows the example after instrumentation.

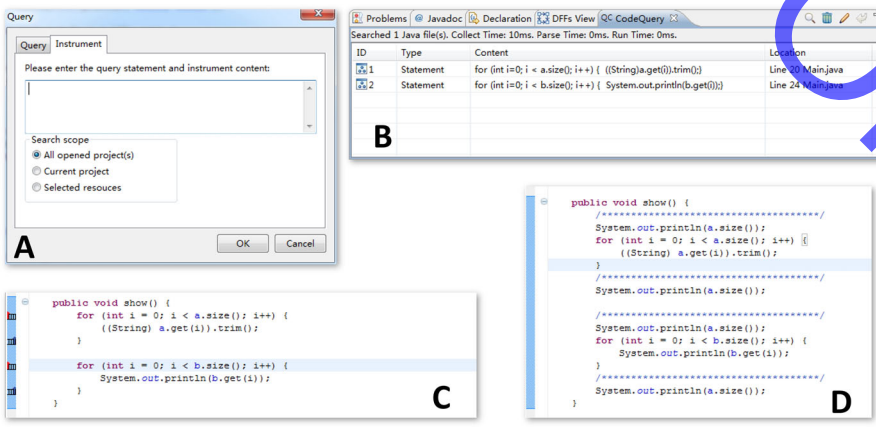



**Fig. 11** Process of source information generation

## 6 Evaluation

This section evaluates the provided prototype tool by comparing some other instrumentation tools widely used, such as BCEL [1] and Soot (Valle-Rai et al. 2000). To illustrate the flexibility, simplicity, and expressiveness of JIns, two typical cases of dynamic analysis are presented by accomplishing the same aims with JIns, BCEL, and Soot. For example, using JIns in the case of AGTCG(Mingsong et al. 2006) comparing to BCEL, and using JIns for JDAMA (Chixiang and Phyllis2011) comparing to Soot.

### 6.1 AGTCG: BCEL versus JIns

AGTCG is a tool for automatic test case generation from UML activity diagrams. By running the program with randomly generated unit test cases, it gets the corresponding program execution traces. Then by comparing these traces with the given activity diagram according to specific coverage criterion, AGTCG tool could reduce the number of test case sets to only containing those meeting the test adequacy criteria. One important part in the core of AGTCG is capturing the execution traces of programs, which is conducted by calling an existing binary (bytecode) instrumentation tool, named BCEL.

**Algorithm 3** Algorithm of AGTCG's instrumenter

**Input:**
*Clist*: A list of Class names in an activity diagram
*Mlist*: A list of Method names in an activity diagram
*Jlist*: A list of JAVA code files
**Output:**
Java code file with instrumentation

```
for each JAVA code file jcf ∈ Jlist do
  while curToken = jcf.nextToken()! = NULL do
    if curToken == "class" then
      curToken = jcf.nextToken();
      if curToken is in the Clist then
        curClass = curToken;
      else
        curClass = NULL;
      end if
    end if
    if curClass! = NULL&curToken ∈ Mlist then
      Insert log.write(curClass, curToken, "begin");
      tempToken = curToken;
      Locate the curToken to the end of the method;
      Insert log.write(curClass, tempToken, "end");
    end if
  end while
end for
```

From an UML activity diagram, AGTCG finds classes and methods to be analyzed, and stores them into several lists. Then, it searches the related byte-code files. When all these conditions are met, it traverses these files one by one to find if these classes or methods are in the lists. If yes, it will insert probes into the byte-code file. At last, it executes the program with instrumentations and obtains the execution traces.

[1] http://jakarta.apache.org/bcel/

The instrumentation module of AGTCG is implemented by calling BCEL APIs. For the reason that BCEL only provides several APIs to dealing with byte-code of Java program, authors have to design their instrumentation algorithms and then implement them by themselves. The Algorithm 3 shows the process of instrumentation with BCEL in AGTCG. Steps 12 and 15 are implemented by calling a series of BCEL APIs. There are a lot of byte-code details in the source code, such as constant pool copy, INVOKEVIRTUAL instruction, ALOAD instruction. In fact, the total of instrumentation code in AGTCG is over 1,400 lines.

In contrast, JIns command could be built to accomplish the same goal instead of BCEL within several sentences. For each method in the MList, two JIns commands can be used instead, defined respectively on Line 1-8 and Line 10–17 in Fig. 12. These two query processes can perform all instrumentations instead of a complex algorithm and hundreds of lines of code when BCEL was used.

As we have discussed in Sect. 4, time consumptions of inserted fragments may disturb the original execution, especially in real-time systems, and may even cause some unexpected results. Though it is not necessary of calculating time cost in the case of AGTCG, still, we present the statistics of time consumptions in Table 3 to show the time effect of the instrumented code.

The values in Table 3 are calculated by running the case study of AGTCG in (Mingsong et al. 2006). There are 14 instrumentation points matched by the JIns query command, and the ID numbers are assigned according the order of their appearance in the source code. After the replacement of BCEL with JIns to implementing the instrumentation module of AGTCG, we test it using 200 test cases provided by AGTCG itself. The 4th test case (0533002, 002007, false, 175, true, 22.43) is chosen because it covers 13 paths among 18 total paths. In the table, values of time consumed are calculated in nanosecond, and the proportion is the average time of the total executing time.

As shown in Table 3, the statistics of time consumptions and their proportion reflect the impact of each inserted code on the time dimension. Most of the instrumented fragments, except No. 6 and No. 14, consume very little time less then 0.1 % of the total running time. Hence, these instrumentations might be ignored even in some real-time systems. While the left inserted fragments, No. 6 and No. 14, cost more time than others, especially No. 14, the maximum time consumption costs over two hundreds times of No. 13, the minimum time consumption. The reason that the executing time of different instrumented code may vary so much lies in the difference of the corresponding instrumentation points. E. Tang and X. Li et al. gave the detailed discussion about it in (Tang et al. 2000).

According to the statistics of the time effect by instrumentations, developers are able to know precisely that on what extend each inserted code disturbs the executing time of the original program. For example, as shown in Table 3, the instrumentation No. 14 consumes on the average 2.09 % of the total executing time. Therefore, the developers should keep vigilant about this point in testing, debugging, and maintenance of the system.

### 6.2 JDAMA: SOOT vs. JIns

JDAMA (Java Database Application Mutation Analyzer) is a mutation testing tool for Java programs that interacts with a database via the JDBC interface. The tool extends the mutation testing approach for SQL provided by Tuya et al. (2006), Javier and de la (2007), through analyzing and dynamically running byte-code programs with instrumentations. The instrumentation module of JDAMA is inherited from BodyTransformer, an abstract

```
find s:statement
satisfying exist m:method exist c:class
where m.name='method'
  && s isIn m
  && m isIn c
insert before s[0]
  'log.write(%c.name%, %m.name%, "begin");';
  calculate

find s:statement
satisfying exist m:method exist c:class
where m.name='method'
  && m.specifyType='return'
  && s isIn m && m isIn
insert after s
  'log.write(%c.name%, %m.name%, "end");';
  calculate
```

**Fig. 12** JIns instrumentation commands for AGTCG

**Table 3** Running time of each instrumentation in AGTCG

| ID | Run 1st (ns) | Run 2nd (ns) | Run 3th (ns) | Run 4th (ns) | Run 5th (ns) | Avg. Time (ns) | Proportion (%) |
|---|---|---|---|---|---|---|---|
| No. 1 | 1812882ns | 592597 | 711778 | 636627 | 413163 | 833409.4 | 0.04 % |
| No. 2 | 446268 | 474409 | 421107 | 78601142 | 354565 | 424087.25 | 0.02 % |
| No. 3 | 336357 | 368800 | 70087615 | 343640 | 953782 | 500644.75 | 0.02 % |
| No. 4 | 401576 | 435012 | 601204 | 435012 | 405217 | 455604.2 | 0.02 % |
| No. 5 | 410514 | 368469 | 359862 | 363503 | 360193 | 372508.2 | 0.02 % |
| No. 6 | 351254 | 11234173 | 343309 | 337349 | 1172282 | 2687673.4 | 0.12 % |
| No. 7 | 90994023 | 5157576 | 342316 | 346288 | 363173 | 1552338.25 | 0.07 % |
| No. 8 | 325432 | 324107 | 426404 | 389988 | 309872 | 355160.6 | 0.02 % |
| No. 9 | 482685 | 3360585 | 343639 | 350261 | 394292 | 986292.4 | 0.04 % |
| No. 10 | 348606 | 7277350 | 361848 | 765410 | 483347 | 1847312.2 | 0.08% |
| No. 11 | 478547 | 489968 | 506852 | 445937 | 471429 | 478546.5 | 0.02% |
| No. 12 | 412169 | 492285 | 344302 | 31029876 | 502879 | 437908.75 | 0.02% |
| No. 13 | 322452 | 318480 | 380388 | 315830 | 303912 | 328212.4 | 0.01% |
| No. 14 | 94293693 | 75868245 | 24175289 | 34836398 | 7858360 | 47406397 | 2.09% |
| TT | 10773674479 | 171276284 | 149953403 | 177807095 | 42740776 | 2263090407 | 100.00% |

[1] *TT Totle time of running instrumented program*

class in Soot, and implements its method internalTranform(). The method internalTranform() is also used to add a method invocation after each query execution point.

In the instrumentation module of JDAMA, the byte-code instructions of target program is traversed, and the instrumentation points are located by the method executeQuery(String queryStr). Then, the context information is collected by some code developed by users. After that, the method MutantChecker(String, int) is called to insert code into the points.

Before re-implementing the instrumentation module using JIns, we illustrated the pattern of instrumentation process in JDAMA in detail as the follows: (1) iterate the bytecode instructions of the target program; (2) retrieve execution points at the signature java.sql.Statement: java.sql.ResultSet executeQuery(java. lang.String), and (3) add an invocation of MutantChecker(String, int) method after each execution point in the class file. This function call has two arguments, i.e., the concrete query being executed and a unique number identifying the execution point. After instrumentation, JDAMA saves the intermediate program files in Jimple (Valle-Rai et al. 2000) format, which could be seen as an extension based on Java source. Figure 13 shows a simple example fragment of code before JDAMA instrumentation, and Fig. 14 shows the case after instrumentation in form of Jimple code.

At first sight, the instrumentation process in JDAMA is very similar to that of JIns, e.g., they both perform instrumentations in the steps of *querying locations*, *generating contextualized code*, and *inserting code to target programs*, but JDAMA leaves the job of programming to users. On the contrary, JIns provides a high level of language and hides away the job of implementations.

To accomplish the same instrumentation goal, a much simpler way of using JIns command could be used instead, as is shown in Fig. 15. It is obviously that, in terms of lines of code, the JIns command is far less then those using Soot APIs in JDAMA.

## 7 Related work

Existing instrumentation techniques can be broadly divided into two categories: instrumentation based on specific points in source programs and instrumentation based on generic instructions in binaries.

***AOP tools are representative of the former*** Tools based on AOP(Kiczales et al. 1997) can insert instrumentations into program according to predefined points. AspectJ [2] is a heavyweight implementation of AOP. Users can complete some instrumentation work by using AspectJ upon predefined crosscutting joinpoints: caller/callee, variable get/set, if branch and scope limiting. These joinpoints provide great convenience for users, but also limit the flexibility in instrumentation points selection. For example, instrumentations at the statement level cannot be always applied in AspectJ. On the other hand, our approach can perform instrumentation at the statement level to offer more flexibility than AOP with respect to instrumentations.

For example, in Fig. 3, two for-statements use java.util.List objects as a part of their loop invariants, and one wants to check if the size of the List objects is changed in the for-statements. The probes must be inserted before and after the for-statements. AspectJ cannot finish this instrumentation work.

In the example, users must find out all the for-statements satisfying the specification. In coverage-oriented testing method, it requires to get the exhaustive execution traces of a program. The instrumentation module must insert probes for if-else branches, loop statements, and some special code blocks. AspectJ cannot implement instrumentation at the statement level.

In addition, JIns is a declarative approach that aims at providing users a high-level language, simple, and expressive, to specify query and instrumentation requirements. By

[2] http://eclipse.org/aspectj/

```
1 public static void findByPrice(Connection con,
2   int x, int y) throws SQLException,
3   Exception {
4   String queryStr;
5   if(x == 0) {
6     queryStr = "SELECT cof_name FROM
7       coffees WHERE price = " +
8       y + "; ";
9   } else {
10    queryStr = "SELECT sup_name, cof_name
11      FROM coffees, suppliers WHERE
12      coffees.sup_id = suppliers.sup_id
13      AND price <= " + y + " ;";
14  }
15  Statement stmt=con.createStatement();
16  ResultSet rs=stmt.
17  executeQuery(queryStr);
18  // ... code that outputs the result set
19 }
```

**Fig. 13** Code fragment to be analyzed in JDAMA

using AspectJ as an instrumentation tool, users mainly work at the programming level, similar to the cases discussed in Sect. 6.

*We can categorize others as Non-AOP tools* They are mainly implemented based on binary (or byte-code) instructions. For example, Valgrind (Nethecote and Seward 2007), Pin (Luk et al. 2005), Nuntia (van der et al. 2007), BCEL, etc. The common feature of these tools is that they all manipulate the instructions of program such that instrumentations can be inserted between instructions. Therefore, these tools are more flexible than others based on source code. Meanwhile, they can reach the minimum adverse influence to time and space performance and do not require the availability of source code. But the feature also causes some disadvantages.

First, binary instrumentation approach cannot focus on instrumentation at the statement level in the source code. For example, it cannot distinguish among for-statement, while-statement and do-while-statement from Java byte-code, because all of these statements are compiled to IF instructions. For the same reason, continue-statement and break-statement cannot be distinguished, either.

Second, users must know the details of program binary/byte-code. For example, BCEL can recount instruction's offset in the process of instrumentation, but programmers still need to know some details of JVM instructions, e.g., the constants pool. Therefore, these tools cannot help programmers locate the instrumentation points at a high level of abstraction

In addition, designing instrumentation programs require familiarity to the APIs provided by the libraries specific to different tools. As a consequence, the instrumentations using such APIs are not succinct. For example, using BCEL to rewrite a field in class requires around 200 lines of extra code, which may extend to over 400 lines in Valgrind.

*SQL-like language is easy-to-use* The JIns Language makes use of predicate logic expression to describe the instrumentation location. The most important issue of the process is how to identify the relationships between instrumentation locations and other program elements. Some other tools also provide such kind of functionality. For example,

```
1  public static void findByPrice
2    (java.sql.Connection, int, int)
3    throws java.sql.SQLException,
4    java.lang.Exception {
5      java.sql.Connection r0;
6      int i0, i1, i2, $i3, $i4, i5, $i6, $i7;
7      java.lang.StringBuilder $r1, $r2, $r3, $r5, $r6, $r7;
8      java.lang.String r4, $r13, $r19;
9      java.sql.Statement $r8;
10     java.sql.ResultSet r9;
11     java.sql.ResultSetMetaData r10;
12     java.lang.StringBuffer $r11, r12, $r17, r18;
13     java.io.PrintStream $r16, $r22;
14     boolean $z0;
15     r0 := @parameter0: java.sql.Connection;
16     i0 := @parameter1: int;
17     i1 := @parameter2: int;
18     if i0 != 0 goto label0;
19     $r1 = new java.lang.StringBuilder;
20     specialinvoke $r1.<java.lang.StringBuilder:\\
21 void <init>(java.lang.String)>(" SELECT cof_name\\
22 FROM COFFEES WHERE price = ");
23     $r2 = virtualinvoke $r1.<java.lang.StringBuilder:
24 java.lang.StringBuilder append(int)>(i1);
25     $r3 = virtualinvoke $r2.<java.lang.StringBuilder:
26 java.lang.StringBuilder append(java.lang.String)>(";");
27     r4 = virtualinvoke $r3.<java.lang.StringBuilder:
28 java.lang.String toString()>();
29     goto label1;
30 label0:
31     $r5 = new java.lang.StringBuilder;
32     specialinvoke $r5.<java.lang.StringBuilder: void
33 <init>(java.lang.String)>("SELECT sup_name, cof_name
34 FROM coffees, suppliers WHERE coffees.sup_id =
35 suppliers.sup_id AND price <= ");
36     $r6 = virtualinvoke $r5.<java.lang.StringBuilder:
37 java.lang.StringBuilder append(int)>(i1);
38     $r7 = virtualinvoke $r6.<java.lang.StringBuilder:
39 java.lang.StringBuilder append(java.lang.String)>(";");
40     r4 = virtualinvoke $r7.<java.lang.StringBuilder:
41 java.lang.String toString()>();
42 label1:
43     $r8 = interfaceinvoke r0.<java.sql.Connection:
44 java.sql.Statement createStatement()>();
45     r9 = interfaceinvoke $r8.<java.sql.Statement:
46 java.sql.ResultSet executeQuery(java.lang.String)>(r4);
47     staticinvoke <instrumenter.Mutation: void
48 MutantChecker(java.lang.String,int)>(r4, 1);
49         //...Jimple code that uses result set
50         return;
51 }
```

**Fig. 14** Jimple code fragment after instrument in JDAMA

```
1 find s:statement
2 satisfying exist m:method
3 where m.name='executeQuery'
4   && m.returnType='java.sql.ResultSet'
5   && m.argumentsType='java.lang.String'
6   && s call m
7 insert after s
8   'instrumenter.Mutation
9     .MutantChecker(%m.arguments[0]%, 1);'
10 ;
```

**Fig. 15** JIns instrumentation commands for JDAMA

CrocoPat (Beyer 2006) uses RML to describe relationships with program and uses RSF(Rigi(Miiller and Klashinsky 1988) standard format) file to store the relationships. However, Crocopat does not perform instrumentations. Since it is a set-oriented calculator, it is not possible to preserve the ordering in the results as required by the code instrumentations.

*XML-based query and transformation languages are instructive* Some other XML-based query and transformation languages are also very popular to programmers, such as XQery (W3C XML Query [3]) and XSLT[4]. They could provide similar declarative source query and instrumentation capabilities based on the XML representation of program ASTs. XQuery is designed by W3C (World Wide Web Consortium) as a query and functional programming language to query collections of XML data. XSLT (Extensible Stylesheet Language Transformations) is another language also developed by W3C for transforming XML documents into other XML documents. As for XML documents, they have done quite well of navigation and manipulation among nodes through XPath. Hence, they are very instructive to developers designing new query languages for well-structured data. However, if we want to use XQuery/XSLT in Java source query and instrumentation, a solid bridge between Java ASTs and XML should be built, including matured APIs and IDEs. Unfortunately, such kind of bridges are still under construction.

Table 4 shows some tools in relation to us. Some of them are not designed for instrumentation, such as BCEL, Soot, and AspectJ. All of these can perform the instrumentation in method level, but most of them can not perform the statement-level instrumentation well. AspectJ and InsECTJ cannot instrument without crosscuttings, therefore, are not suitable to loop statements. The typical application of this tools is API programming. By using BCEL and Soot, users can read instrumented binary code that, however, is hard to be comprehended.

Another category of tools and methods are designed for code query (Alves et al. 2011), such as CPPX/FETCH(Du et al. 2007), CrocoPat, Grok (Holt 2008), and TXL (Cordy 2006). CPPX/FETCH is a fact extractor, which can be piped to Crocopat/Grok to query about the code using the Rigi standard format. Crocopat and Grok can compute regular expressions as well as transitive closures. In contrast, TXL is a transformation language which can be applied to instrumentation, though it requires language grammar programming that are not trivial for end-programmers to use.

[3] http://http://www.w3.org/

[4] http://www.w3.org/TR/xslt20/

**Table 4** Comparison of some common instrumentation tools

| | Designed for Instrumentation | Method Level | Statement Level | Instruction Level | User Interface | Readability | Instrumentation Mode |
|---|---|---|---|---|---|---|---|
| Valgrind(Nethecote and Seward 2007) | Yes | Yes | No | Yes | API Programming | No | Dynamic |
| Pin (Luk et al. 2005) | Yes | Yes | No | Yes | API Programming | No | Dynamic |
| BCEL | No | Yes | No | Yes | API Programming | Partly | Static |
| Soot (Valle-Rai et al. 2000) | No | Yes | Yes | No | API Programming | Partly | Static |
| AspectJ | No | Yes | Partly | No | AspectJ Language | No | Dynamic |
| InsECTJ (Seesing and Orso 2005) | Yes | Yes | Partly | No | API Programming | No | Dynamic |
| CPPX/FETCH (Du et al. 2007) | No | Yes | Yes | No | SQL Query | Yes | Static |
| CrocoPat (Beyer 2006) | No | No | No | No | Relation Query | Yes | |
| Grok (Holt 2008) | No | No | No | No | Grok Query | Yes | |
| TXL (Cordy 2006) | No | Yes | Yes | No | Grammar Query | No | Static |
| JIns | Yes | Yes | Yes | No | JIns Command | Yes | Static |

Most existing instrument tools dedicate themselves in providing more powerful and effective API to complete instrumentation. However, they ignore how to declaratively describe instrumentation points and constructing instrumentation statements. This situation requires programmers to go through hundreds of pages to perform even a small instrumentation task. New program errors are likely to be caused by such programming tasks. By the design, JIns is a simple, flexible, and effective instrumentation interface to end-programmers.

## 8 Conclusions and future work

We have presented an implemented instrumentation approach based on a new code query technology for Java programs. The tool solves two related issues: to query the instrumentation points declaratively and to instantiate the instrumentation templates using the query results. The JIns language is designed for fine-grained and context-sensitive instrumentation tasks. Through a streamlined Eclipse tool support, the complexity of code instrumenting can be greatly reduced. As indicated by two nontrivial practical case studies, our framework has been evaluated against the widely used instrumentation frameworks such as BCEL and SOOT, achieving significant savings in terms of lines of specifications.

Our future work is planned as follows: first, we plan to support more programming languages, such as C/C++. Also, we plan to provide more instrumentation templates for programmers to further reduce adverse influence of time and space performance.

**Acknowledgments** This work is supported by the National Natural Science Foundation of China (No. 61003025, No. 61021062) and the National 863 High-Tech Programme of China (No. 2011AA010103, No. 2012AA011205).

## References

AspectJ project website http://eclipse.org/aspectj/.

Alves, T. L., Hage, J., & Peter (2011) Rademaker: A comparative study of code query technologies. In: *Proceedings of the 2011 IEEE 11th International Working Conference on Source Code Analysis and Manipulation*(SCAM'11), (pp. 145–154), IEEE Computer Society, 2011.

BCEL (Byte-Code Engineering Library) website, http://jakarta.apache.org/bcel/.

Beyer, D. (2006). Relational programming with CrocoPat, In *Proceedings of the 28th international conference on Software engineering* (ICSE '06), (pp. 807–810).

Chixiang, Z.,& Phyllis, F. (2011). JDAMA: Java database application mutation analyser. *Software Testing, Verification and Reliability, 21*(3), 241263, Wiley, 2011.

Cordy, J. R. (2006). The TXL source transformation language. *Science of Computer Programming, 61*(3), 190–210.

Du Bois, B., Van Rompaey, B., Meijfroidt, K., & Suijs, E. (2007). Supporting reengineering scenarios with FETCH: an experience report. In *Proc. of the 3rd International ERCIM Workshop on Software Evolution*, ICSM, p.69C82, Paris, France, October 2007.

Holt R. C. (2008). Grokking software architecture. In: Proceedings of the 15th Working Conference on Reverse Engineering (WCRE'08), pp 5–14. IEEE, 2008. Most influential paper.

Javier, T. Cabal, M. J. S., & de la Riva, C. (2007). Mutating Database Queries. *Information and Software Technology, 49*(4), 398–417.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., et al. (1997). Aspect-oriented programming, In *Proceedings of the European Conference on Object-Oriented Programming*(ECOOP), Springer LNCS 1241. June 1997.

Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., et al. (2005). Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*(PLDI2005), (pp. 190–200).

Mingsong, Chen, Xiaokang, Qiu, & Xuandong, Li (2006). Automatic Test Case Generation for UML Activity Diagrams, In *Proceedings of the 2006 international workshop on Automation of software test*, 23 May 2006, (pp. 2–8). New York: ACM Press.

Miiller, H. A., & Klashinsky, K. (1988). Rigi-A system for programming-in-the-large, In *Proceedings of the 10th international conference on Software engineering*(ICSE'88), Raffles City, Singapore Aprid 11–15, (pp. 80–86), April 1988.

Nethecote, N. & Seward, U. (2007). Valgrind: reducing maintenance effort through software operational knowledge: An eclectic empirical evaluation. In: *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*(PLDI2007), (pp. 89–100).

Seesing, A., & Orso, A., (2005). InsECTJ: A generic instrumentation framework for collecting dynamic information within Eclipse[C]. In Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange(eclipse2005), pp. 45–49.

Tang, E., Wang, L., Zhao, J., Li, X., Wang, L. Zhao, J., et al. (2012) : Time-leverage point detection for time sensitive software maintenance, In *Proceedings of the 28th IEEE International Conference on Software Maintenance*, (pp. 567–570), IEEE Computer Society.

Tuya, J., Cabal, M. J. S. & de la Riva, C. (2006) SQLMutation: A tool to generate mutants of SQL database queries. In *Second Workshop on Mutation Analysis (Mutation 2006 - ISSRE Workshops 2006)*.

van der Schuur, H., Jansen, S., & Brinkkemper, S. (2007). Reducing maintenance effort through software operation knowledge: An Eclectic Empirical Evaluation. In: *Proceedings of the 15th European Conference on software maintenance and reengineering*(CSMR2011), IEEE Computer Society, (pp. 201–210), 2007.

Verbaere, M., Godfrey, M. W., & Girba, T. (2008). Query technologies and applications for program comprehension[C]. In *16th IEEE International Conference on Program Comprehension*(ICPC), pp. 285–288.

Valle-Rai, R., Co, P., Gagnon, E., Hendren L., Lam, P., Sundaresan, V. Soot: a Java bytecode optimization framework. In: *In CASCON First Decade High Impact Papers*(CASCON '10), pp. 214–224, IBM Corp., NJ, USA.

Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D. et al. (2008). The worst-case execution-time problemoverview of methods and survey of tools, *ACM Trans. Embedd. Comput. Syst.*, 7(3) (pp. 1–53). New York: ACM.

World Wide Web Consortium (W3C) homepage. http://http://www.w3.org/.

W3C XML Query (XQuery). http://www.w3.org/XML/Query/.

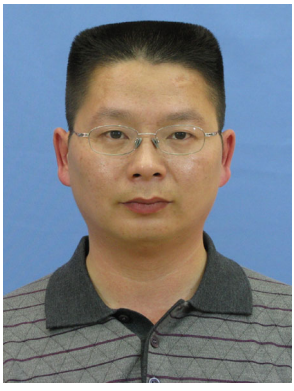
XSL Transformations (XSLT).

## Author Biographies

**Tian Zhang** is an associate professor at the Computer Science and Technology Department of Nanjing University, China. At the same time, he is also a research fellow at the State Key Laboratory for Novel Software Technology. He obtained the permanent position in Nanjing University after graduation. His overall research interests relate to model driven aspects of software engineering, with the aim of facilitating the rapid and reliable development and maintenance of both large and small software systems. Currently, he is in charge of a National Natural Science Foundation of China (No.61003025) and a Jiangsu Province Research Foundation (BK2010170), both related to real-time and embedded systems.

**Xiaomei ZHENG** received her master degree from China University of Mining and Technology in 2004. Currently she is a lecture at the Information Technology School of Nanjing University of Chinese Medicine. Her research interests include model-driven development and model-driven tools to improve quality and efficiency in software development processes.

**Yan ZHANG** is a lecture at Beijing Electronic Science and Technology Institute. He received his MS and PhD degrees in Computer Science from University of Science and Technology Beijing and Nanjing University, respectively. He received his BS degree in Computer Application from Beijing Union University. His research focuses on Component Based Software Engineering (CBSE), Cyber-Physical System (CPS) and modeling and verification of software systems.



**Jianhua ZHAO** is a professor at the Computer Science and Technology Department of Nanjing University. He received his MS and PhD degrees in Computer Science from Nanjing University, China, in 1996 and 1999 respectively. His research interests include model checking, data purification technology, software engineering, programming language.

**Xuandong Li** received his MS and PhD degrees from Nanjing University, China, in 1991 and 1994, respectively. He is a full professor at the Computer Science and Technology Department of Nanjing University. Currently, he is the dean of both the Computer Science and Technology Department and Software Institute of Nanjing University. His research interests include formal support for design and analysis of reactive, disturbed, real-time, hybrid, and cyber-physical systems; software testing and verification; model driven software development and service oriented computing.