A Pointcut Language for Setting Advanced Breakpoints

Haihan Yin, Christoph Bockisch, Mehmet Akşit
Software Engineering group, University of Twente, 7500 AE Enschede, the Netherlands
{h.yin, c.m.bockisch, m.aksit}@cs.utwente.nl

ABSTRACT

In interactive debugging, it is an essential task to set breakpoints specifying where a program should be suspended at runtime to allow interaction. A debugging session may use multiple logically related breakpoints so that the sequence of their (de)activations leads to the expected suspension with the least irrelevant suspensions. A (de)activation is sometimes decided by some runtime context values related to that breakpoint. However, existing breakpoints, which are mainly based on line locations, are not expressive enough to describe the logic and the collaboration. Programmers have to manually perform some repeated tasks, thus debugging efficiency is decreased.

In this paper, we identify five frequently encountered debugging scenarios that require to use multiple breakpoints. For such scenarios, it is often easier than using the traditional debugger to write pointcuts in an aspect-oriented language, and to suspend the execution at the selected join points. However, existing languages cannot handle the scenarios neatly and uniformly. Therefore, we design and implement a breakpoint language that uses pointcuts to select suspension times in the program. Our language allows programmers to use comprehensible source-level abstractions to define breakpoints. Also, multiple breakpoints can be freely composed to express their collaboration. In this way, an expected suspension can be expressively programmed and reached with less or even no irrelevant suspensions.

Categories and Subject Descriptors

D.2.5 [**Testing and Debugging**]: Debugging aids; D.3.2 [**Language Classifications**]: Very high-level languages; D.2.2 [**Design Tools and Techniques**]: User interfaces

Keywords

Debugging, Advanced breakpoint, Pointcut language

1. INTRODUCTION

Software is kept being maintained from its delivery until its end. Maintenance takes the majority of effort spent in developing software and a significant portion of maintenance is carried out for debugging [14, 11]. An important step, which is called fault localization, of debugging is finding out the root cause based on some observed symptoms. The root cause always happens before unexpected symptoms appear. Eisenstadt [9] studied 59 bug anecdotes and he concluded that over 50% of the difficulties resulted from this temporal or spatial chasm between the root cause and the symptom, or from inapplicable debugging tools.

In interactive debugging, programmers use breakpoints to mark places in the source code where the program should be suspended at runtime. When the debuggee program is suspended, programmers can inspect the program state, observe the program behavior, or perform other debugging tasks by using a debugger. How breakpoints are set can significantly affect the efficiency of debugging. An inexperienced programmer may set too many breakpoints; redundant ones distract her attention from those revealing the root cause. Or she may set too few, which results in passing the root cause.

Breakpoints should not be arbitrarily set, because each suspension has a cost. At least, programmers need to decide whether the suspension is relevant. Observing a symptom, programmers usually first roughly choose a program slice that is most likely to cause the symptom, and then set breakpoints to observe the slice. For example, if a field stores a wrong value, breakpoints are set at places where the field is modified. Thus, logically, breakpoints are grouped by programmers according to what they do instead of where they are. However, the traditional breakpoints are mainly based on source lines, which do not embrace any logic of why breakpoints are placed there [7]. Programmers have to mentally sketch the logic relationships between these breakpoints.

Also, breakpoints are independent from each other. At runtime, each breakpoint has its states such as being activated, and contexts such as the value of a variable. Sometimes, a desired suspension requires multiple breakpoints and their states which sequentially form a path leading to the suspension. This may require non-trivial manual effort, such as recording the state or context at one suspension and using it at another. Limitations of the traditional breakpoints often require programmers to perform many repeated debugging steps. Thus, debugging efficiency is decreased.

We identify five frequently encountered scenarios that the traditional breakpoints cannot handle well. The identified scenarios require breakpoints to be set at places sharing some common characteristics, such as similar syntax. The concept of *pointcut* perfectly fits in this context. These scenarios show that using pointcut-advices to construct places for setting breakpoints is a more convenient and efficient approach than traditional debugging. However, current AOP languages are not specifically designed for solving these scenarios. Thus, pointcut-advices are too verbose. Furthermore, pointcut-advices cannot be used to set a breakpoint to specific advice compositions, which is one of our identified scenarios. Also, programs that are added for debugging may accidentally stay in the project. This will introduce unnecessary maintenance effort in the future.

Therefore, we propose a declarative breakpoint language (BPL). By building on AspectJ, BPL can be used to debug Java or AspectJ programs. The breakpoint, that is the core concept of BPL, is a first-class value. A breakpoint can be defined by AspectJ-like pointcuts which use source-level abstractions. This makes the description of breakpoints more comprehensible than line breakpoints. We extend and improve existing AspectJ pointcut designators with seven novel ones. In BPL, breakpoints are named and can be used to compose higher-level breakpoints. The composition level can be infinite because we treat the primitive and the composed breakpoints in a uniform way. BPL is the first approach providing a pointcut for selecting a specific action composition at runtime.

This paper is structured as follows. Section 2 presents five debugging scenarios and describes how debugging processes are performed in different approaches. Section 3 gives a detailed introduction to the new features introduced in BPL. Section 4 highlights several implementation considerations in our prototype. Section 5 describes two debugging examples by using the traditional debugging, the program solution, and our solution respectively. Section 6 and 7 describe related work and conclude this paper respectively.

2. PROBLEM STATEMENTS

Debugging is a cognitive process and how it is performed significantly depends on the programmer's observations and experience. A programmer tends to give the same treatment when she observes the same symptom, such as a certain exception being thrown. In this section, we select several debugging scenarios that are frequently encountered. For each scenario, we elaborate the way of using the traditional debugger. We tag debugging steps in the description like "a.1", in which the letter represents a debugging process and the digit represents the step order of that process.

Each scenario requires multiple breakpoints or suspensions at different locations, which share some common properties, such as similar syntax, relation to the same variable, etc. In AspectJ, a pointcut is used to select places with common properties. This has inspired us to use AspectJ programs during debugging, where pointcuts select the join points at which we want to suspend the execution and where we set a breakpoint in the otherwise empty advice body. In this section, we also demonstrate this approach for the identified scenarios. Actually, this approach is a variant of program instrumentation.

The program solution serves two purposes. First, the program can describe the scenario in a more succinct and clear way than instructions for manual debugging given in natural language. Take pointcut **call**(**void** Shape.set∗(..)) for example, it can be seen as two debugging tasks in this context: finding all places calling methods which satisfy the pattern "void Shape.set∗(..)", and then setting line breakpoints there. Second, the programs will be compared with our solution, which is an AspectJ-like language.

We have two basic criteria for the program solution. First, the program should be in a separate module. AspectJ modularizes scattering concerns and we want to keep this principle in the program solution. Second, the program should be simple. Effort spent on writing the program should be comparable to that spent on the traditional debugger. In most cases, writing a lengthy or a sophisticated program for setting a breakpoint is not desirable.

2.1 Scenario 1: Selecting Multiple Locations

Sometimes, it is difficult to decide which specific location is executed at runtime. For example, to debug unexpected behavior in a system, which the programmer is not familiar with, she may deduce roughly which function is executed by matching names of the function with the observed runtime behavior. A function can be implemented as a set of overloading constructors or methods. However, to know which specific one is executed at runtime, she may need to set a breakpoint to each implementation.

The difficulties of using the traditional debugging in this scenario mainly come from finding locations for setting breakpoints and managing breakpoints as logic units.

Finding locations.

Suppose the programmer observes that a field stores an unexpected value, she needs to monitor the runtime states of this fields. The first option is setting a watchpoint to this field (**a.1**). When the watchpoint is hit, the programmer needs to perform one "step over" to inspect the field value after the modification (**a.2**).

The second option requires the programmer to manually find out the last assignment (**b.1**) to this field before the unexpected value is observed. This assignment can be in any constructor or method modifying this field. She needs to set a line breakpoint to each found place (**b.2**) and specify a condition to check whether the expression on the right-hand side of the assignment equals the unexpected value (**b.3**).

Managing breakpoints.

Using the *Breakpoint* view provided in modern IDEs, such as Eclipse, the programmer can organize the breakpoints she set. The view can group the breakpoints according to their types, such as line breakpoint or watchpoint, or their locations, such as files or projects. Breakpoints can be (de)activated and deleted at the granularity of groups. However, there is no approach provided to group breakpoints as logic units. Thus, a debugging task applied to a logic operation will possibly required repeating steps.

Listing 1 shows how an AspectJ program monitors unexpected assignments to the field Clazz.var. AspectJ can access the value assigned to a field by using **args**(). The pointcut describes the desired places for suspensions and a breakpoint is set in the body of the **before** advice. When the program is suspended in the advice, the programmer can locate the root cause by using the stack trace. Usually, the second top frame in the stack trace points to the root cause, because the top frame represents the execution of the advice.

```
public aspect Scenario1Aspect {
  before(int val) : set(int Clazz.var) && args(val) &&
      if(val==/*Unexpected value*/) {
    // set a breakpoint on this line
  }
}
```

Listing 1: An AspectJ program monitoring assignments to a field

2.2 Scenario 2: Monitoring Updates on a Field

Listing 2 shows a program slice updating a field, which is a HashMap. On lines 4-8, we use ellipsis to indicate that the separated statements may reside in different methods and their execution order is not the same as the lexical order.

```
1 class Scenario2 {
2   private HashMap map1;
3
4   map1.put(key1, value1);
5   ...
6   map1.get(key2);   //returns a null value
7   ...
8   map1.put(key2, value2);
9 }
```

Listing 2: Multiple places updating a field

When a value retrieved from the HashMap is wrong, as line 6 shows, the potential root causes are places updating this HashMap, such as lines 4 and 8.

To debug this with a traditional debugger, the programmer needs to find all the updating locations (**c.1**), set breakpoints there (**c.2**), and evaluate the values of the expressions for updates at runtime (**c.3**). A watchpoint for the field is not helpful in this scenario, because it can only suspend the program when the field is accessed or modified instead of being updated. Setting a breakpoint to the called method HashMap.put() may result in redundant suspensions, because there may be other HashMaps in the program.

Listing 3 gives an AspectJ solution for this scenario. It is a *privileged* aspect which can access protected members of other classes. On line 6, the advice checks whether the callee object t is same as the value stored in the expected field, such as the private field map1 in Listing 2.

```
1 public privileged aspect Scenario2Aspect {
2   before(Scenario2 caller, HashMap t, String s) :
3       call(public Object HashMap.put(..)) && this(caller) &&
4       target(t) && args(s, *) &&
5       if(s.equals(/*value of key2*/)) {
6     if(caller.map1 == t) {
7       // set a breakpoint on this line
8     }
9   }}
```

Listing 3: An AspectJ program monitoring updates on the object referenced by a specific field

2.3 Scenario 3: Finding Null Pointer Dereferences

The dot operator dereferences an object pointer to access a member from that object. A line of code may contain multiple dereference operations as in the following listing:

```
total.getObjects().addAll(current.getObjects());
```

If a *NullPointerException* occurs on this line, the error message only tells the line number where the exception occurs instead of the specific operation. For debugging this scenario, the programmer needs to place a breakpoint at the line where the exception occurs (**d.1**). When the program is suspended, she needs to repeatedly perform "step into" and then "step return" to check each dereference operation until the exception occurs (**d.2**). Meanwhile, she has to manually note which dereference operation the debugger reaches (**d.3**).

Another option is to change the layout of the code so that there is a dereference operation per line, like the following listing shows (**e.1**). After rerunning the program (**e.2**), the error message can accurately tell which line throws the exception. Since this option requires rewriting the source code, it is also not generally applicable.

```
total.getObjects()
  .addAll(
    current.getObjects());
```

Listing 4 presents an AspectJ program corresponding to this scenario. The pointcut is only satisfied if the receiver of a dereference operation is **null** (see line 5). The advice body further restricts the line number on line 7. If a breakpoint is set at line 8, it can suspend the execution before the dereference operation, which ends up with a *NullPointerException*, is about to occur.

```
1  public aspect Scenarios3Aspect {
2    before(Object receiver) :
3        (call(* *.*(..)) || get(* *.*)) && target(receiver) &&
4        withincode(/*a method pattern*/)
5        && if(receiver == null) {
6      int line = thisJoinPoint.getSourceLocation().getLine();
7      if(line == /*expectedLine*/) {
8        // set a breakpoint on this line
9      }
10   }
11 }
```

Listing 4: An AspectJ program checking *null* receivers on a source line

2.4 Scenario 4 : Recording Execution History

Listing 5 shows program slices related to operations on two stream objects. An exception would be thrown when line 6 is executed, because it tries to read data from a closed Stream.

```
1 InputStream s1 = new FileInputStream(...);
2 InputStream s2 = new FileInputStream(...);
3 s1.close();
4 s2.read();
5 s2.close();
6 s1.read();  // An exception is thrown.
```

Listing 5: A program performing operations on stream objects

An execution path can lead to unexpected behavior, such as first close then read. Programmers need to track the cause backwards from the point where the symptom is observed. However, most traditional debuggers do not provide backtracking. Using breakpoints, the programmer is likely to suspend the program either before or after the cause. If the cause is passed, the programmer needs to restart a new debugging session. Moreover, debugging Listing 5 requires that all events of the path refer to the same object. The programmer has to manually note corresponding information with the traditional debugger.

Tracematch [1] is an AspectJ extension designed for observing execution traces. Therefore, we choose Tracematch as the alternative debugging solution for this scenario. Listing 6 shows a Tracematch program. Lines 2 and 3 define two events named close and read. Both of them bind the **target** value to the parameter of the tracematch (line 1). Events with different **target**s are not recorded by the same **tracematch** instance. Line 4 declares the expected, but undesired execution path with the two names. When the path is matched on the same Stream, the instruction represented by line 5 is executed.

```
1 tracematch(Stream s) {
2   sym close before : call(∗ Stream.close(..)) && target(s);
3   sym read before : call(∗ Stream.read(..)) && target(s);
4   close read {
5     // set a breakpoint on this line
6   }
7 }
```

Listing 6: A tracematch specifying an undesired execution path

2.5 Scenario 5: Exploring a Program Composition

The execution of an advice can alter the flow of its base program to any extent. Many works [17, 8, 13, 15, 12] have identified the problem of aspect (or advice) interference. An incorrect composition, which can be either between advices or between advices and the base program, at a join point causes unexpected runtime behavior. Therefore, the programmer needs to inspect the execution of the composition at such a join point. What complicates this task is that pointcuts can include dynamic tests. Thus, when advice share a join point shadow, these advice are not necessarily executed together.

To debug this scenario, the programmer first needs to find join point shadows (JPS) affected by all advices of the expected composition (**f.1**) and then set breakpoints to these shared JPSs (**f.2**). Because in AspectJ, whether an advice is applied can only be seen when it is actually executed, the programmer needs to execute the program once (**f.3**), manually perform bookkeeping of the program composition at each join point (**f.4**) and record the hit count of the line which contains the desired join point (**f.5**). In the second debugging session, setting line breakpoints with the recorded hit counts (**f.6**) leads to suspensions at the expected join point before any advice is executed.

For this case, AspectJ cannot provide a clean way for putting debugging code in a separate module. There is no pointcut which can uniquely identify the execution of an advice, because advices are unnamed in AspectJ. Furthermore, an advice using the pointcut **adviceexecution**() cannot easily obtain information about the join point triggering the execution of the advice. Without this information, it is impossible to know whether different advice executions are composed at the same join point. Though there are works, such as Oarta [16] and dependent advice [5], supporting named advices, none of them can use advice names to specify an expected runtime composition.

2.6 Summary

In this section, we have described five debugging scenarios that require non-trivial manual tasks such as setting breakpoints, repeating steps, and recording past states. For these scenarios, the program solutions that describe the suspension conditions in a declarative way show their strength and potential.

However, some debugging programs are verbose. In Listing 3, comparing the field value and the current target object is always required in debugging scenario 2. In Listing 4, most of the parts are generic except the location information. These programs can be more reusable if the configurable parts are parameterized. Besides, there is no solution that treats these scenarios in a uniform way. Take scenarios 4 and 5 for example, Tracematch can easily specify a sequential execution of operations a and b. However, it is impossible to reuse the previous declarations to express that the operation a should also be advised by the advice c.

Last but not least, we do not encourage to add code, which is not part of the main functionality, to the source program. The added code may introduce unnecessary maintenance effort if the programmer forgets to remove it after fixing a bug. Even though source control management systems, such as subversion, can be used to tell the differences between two versions, programmers need to distinguish the added debugging programs and the fixed parts.

3. BREAKPOINT LANGUAGE

Based on the observations described in section 2, we have designed and developed a breakpoint language (BPL) for setting advanced breakpoints. BPL reuses many features of AspectJ and Tracematch. Additionally, it has its own unique functionalities.

The debuggee programs of BPL are Java programs or AspectJ programs. During the debugging, breakpoints specified by the BPL suspend the debuggee program at join points where they are satisfied.

Listing 7 shows the grammar rule of a breakpoint declaration. A breakpoint declaration has a name, a parameter list, and a pointcut expression. The rule for PointcutExpr extends the AspectJ pointcut with seven designators. We describe these designators and their usages in the following subsections.

```
BreakpointDeclaration :
  Name '(' FormalParameterList? ')' ':' PointcutExpr ';' ;
```

Listing 7: The grammar rule of a breakpoint declaration

3.1 The Pointcut call()on()

The pointcut **call**()**on**() is derived from the pointcut **call**(). It matches join points where a method is called on an object referenced by a specific field. The **call**() and **on**() parts take the method and the field specifications respectively. This pointcut can be used at any place where **call**() is applicable. It should be noted that the **on**() part matches based on the referential identity of the values, i.e., it also matches alias of the specified field.

Listing 8 shows an example of using **call**()**on**(). Compared to Listing 3 in **Scenario 2**, it implicitly constrains the callee object of the method.

```
bp(String s) :
  call(public Object HashMap.put(..))on(Scenario2.map1) &&
  args(s, ∗) && if(s.equals("key2"));
```

Listing 8: A breakpoint declaration using pointcut call()on()

3.2 The Pointcuts location() and checkNPE()

As the following listing shows, the pointcut **location**() takes three parameters which represent the file path, the file name, and the line number in the file respectively. The third parameter takes a list of line numbers or line ranges, e.g., [97, 100..102]. This pointcut can be used jointly with other pointcuts to restrict locations of JPSs, for example **call**(...) && **location**(...).

```
bp() : location(‘‘Spacewar", ‘‘SpaceObject.java", [97]);
```

The pointcut **checkNPE**() matches dereference operations where the receiver is *null*. A breakpoint using **checkNPE**() suspends the program just before the satisfied dereference operation is performed, and thus the suspension happens before a *NullPointerException* is thrown. The first breakpoint declaration shown in Listing 9 checks whether a line contains a null pointer dereference. The second breakpoint declaration does the same for the specified method body.

Compared to Listing 4 in **Scenario 3**, **checkNPE**() omits the fixed parts specifying the cause of *NullPointerException*; only the source location is left to be configured.

```
bp1() : checkNPE() &&
        location("Spacewar", "SpaceObject.java", [97]);
bp2() : checkNPE() && withincode(/∗a method pattern∗/);
```

Listing 9: Breakpoint declarations using pointcuts location() and checkNPE()

3.3 The Pointcuts path() and bind()

The pointcut **path**() matches a specific execution path existing in the history. It takes a path expression, which consists of breakpoint references, as the parameter. We use a blank space to represent the sequential order and rectangular brackets to represent the exact expected hit count. For example, **path**(a[2] c) expects that breakpoints a and c are hit in the sequence "aac". Expression c is shortened from c[1]. Besides, the "+" sign, which means 1 or more, and "*", which means 0 or more, can be appended to a breakpoint reference. The **path**() expression is satisfied when all referenced breakpoints are satisfied in the sequence specified as the path expression.

The pointcut **bind**() is used to bind context values exposed by lower-level breakpoints to the higher-level breakpoint. Listing 10 shows our solution for **Scenario 4**, which is about recording execution history.

Lines 1 and 2 declare two breakpoints for read and close operations respectively. Lines 3–5 declare a composite breakpoint. Line 4 describes an expected execution path which requires that breakpoints close and read are hit sequentially. Line 5 binds values from lower-level breakpoints to the parameter declared on line 3. A binding relies on the name of a parameter in the composite breakpoint and the position of a parameter in the lower-level breakpoint. For example, read(s) binds the first parameter of the breakpoint read to the parameter named s of the composite breakpoint closeRead. Wildcards can be used to skip parameters that are not relevant to the breakpoint declarations. For example, read(∗, s) and read(.., s) bind the second and the last parameter respectively. In Listing 10, both bindings bind values to the same parameter s and this implies that the bound values must refer to the same object.

```
1 read(Stream t) : call(public ∗ Stream.read()) && target(t);
2 close(Stream t) : call(public ∗ Stream.close()) && target(t);
3 closeRead(Stream s) :
4   path(close read) &&
5   bind(read(s), close(s));
```

Listing 10: Declaration of a composite breakpoint

It is also possible to bind values to different parameters, as Listing 11 shows. The equality of bound values is specified explicitly in the **if**() expression on line 4. Both breakpoints closeRead and closeRead_if suspend the program at the same times. The former is more succinct and the latter is more flexible with restricting the bound values.

```
1 closeRead_if(Stream rStream, Stream cStream) :
2   path(close read) &&
3   bind(read(rStream), close(cStream)) &&
4   if(cStream == rStream);
```

Listing 11: A composite breakpoint using if()

Our solution for the path expression is greatly inspired by Tracematch, but there are two fundamental distinctions. In the view of the structure, a primitive event declared in one tracematch cannot be referred to by other tracematches. In BPL, primitive breakpoints are more reusable, because they can be referred in any number of composite breakpoints. In the view of the join point model, Tracematch is interested in event kinds, such as *before* and *around*. BPL runs with an interactive debugger that provides only forward execution. Therefore, it only suspends the program *before* executions of the satisfied join points.

3.4 The Pointcuts adviceexecution() and composition()

In AspectJ, **adviceexecution**() does not take any parameter and it cannot select the executions of a specific advice. BPL provides a backwards-compatible extension of **adviceexecution**(), which can take the fully qualified name of an advice as the parameter. As an example, pointcut **adviceexecution**(GameInfo.guiInitiation) selects the execution of the advice declared in the aspect "GameInfo" and named with "guiInitiation". Section 4.3 describes how advices are named.

We use the term "action" to refer to an advice, a method or constructor call, a field access, etc. The **composition**() pointcut designator selects join points with an action composition where actions have the specified relationship. To use this pointcut, the programmer first needs to declare breakpoints that suspend the program at the executions of the desired actions. Then, she can use the names of the declared breakpoints to specify a *composition pattern*. Last, the pattern is used as the parameter of **composition**().

We provide two types of composition pattern. Suppose beforeExe and afterExe are two breakpoints that both use **adviceexecution**(). A breakpoint using **composition**() suspends the program at a join point where the composition satisfies the specified pattern.

Existence - the actions referenced in the specified pattern should exist in the composition. We use commas to list breakpoints corresponding to desired actions, e.g., **composition**(afterExe, beforeExe).

Exclusion - the actions referenced in the pattern should not occur in the composition. We use an exclamation mark for this relationship, e.g., **composition**(!afterExe).

4. IMPLEMENTATION CONSIDERATIONS

In earlier work [19], we have developed a debugger for AO programs on top of the execution environment NOIRIn from the ALIA4J language-implementation architecture [3]. In ALIA4J and thus in NOIRIn, aspect-oriented concepts, such as join point and pointcut evaluation, are modelled as first class objects. The AO debugger complements a Java debugger with functionalities for debugging AO features. It allows programmers to inspect the context values, the composition, etc., at a join point. We modified the AspectBench compiler [2] to generate an intermediate representation of AspectJ programs as required by the ALIA4J approach, which preserves the full source locations of AO entities and makes them accessible at runtime.

BPL is implemented to work together with the AO debugger. When a breakpoint is hit at a join point, the programmer can use the AO debugger to observe the suspended program.

At runtime, the breakpoint declarations are sent to NOIRIn. They are evaluated in the context at a join point along with the execution of rest of the program.

4.1 Evaluation of Breakpoints

Figure 1 shows a diagram of the classes that are used in our implementation to represent breakpoints in BPL in the execution environment. *AdvancedBreakpoint* represents the breakpoint and it is managed by a *BreakpointManager*. Each breakpoint has a *Condition* specifying in what condition the breakpoint can be hit. The figure includes only the conditions related to the pointcut designators introduced in section 3.

When the program reaches a join point at runtime, NOIRIn first analyzes the call context, then computes the action composition performed at this join point, and finally executes actions in the composition. Breakpoints are evaluated when an action is about to be executed after all context information, including the call context, the composition, and the executing action, is prepared.

For all primitive conditions except *AdviceExecutionCondition*, it is enough to be evaluated once at a join point. To distinguish this different evaluation frequency, we put a flag to *Condition* and its subtypes. The flag has two values, which are composition-level (*c-level*) and action-level (*a-level*). A breakpoint with a c-level condition is evaluated once at a join point and one with an a-level condition is evaluated at every action in the composition. A binary condition such as *AndCondition* is c-level if and only if its two operators are c-level.

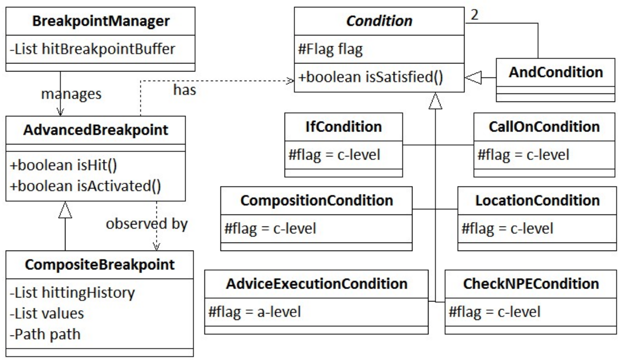


Figure 1: A class diagram of classes related to the breakpoint in BPL.

Multiple breakpoints may suspend the program at the same join point. We use a buffer *hitBreakpointBuffer* to store the hit breakpoints at a join point. A hit breakpoint sends a message with required debugging information to the buffer. When the evaluations of all breakpoints are finished, the *BreakpointManager* checks whether there is any message in the buffer. If the buffer is not empty, the manager emits a suspending request and releases all messages stored in the buffer. The buffer is cleared when the next evaluation process starts.

4.2 Evaluation of Composite Breakpoints

Using a **path**() expression, a composite breakpoint can be composed of primitive breakpoints or other composite breakpoints. It may use **bind**() to access values from lower-level breakpoints and further restrict its suspending condition by using **if**().

In Figure 1, there are two lists in class CompositeBreakpoint. The list *hittingHistory* records the hit history of the lower-level breakpoints. The list *values* records values bound to the parameters. A *CompositeBreakpoint* is an observer of all its lower-level breakpoints. Whenever one of its lower-level breakpoints hits, the evaluation of the composite breakpoint starts. The evaluation first updates lists *hittingHistory* and *values*. Then, it explores the *hittingHistory* list and tries to find an expected path. If an expected path exists, the composite breakpoint starts to evaluate the *if()* condition. If the *if()* condition is true, the composite breakpoint is hit and it produces a message containing all the debugging information, such as locations and bound values, of its lower-level breakpoints.

For illustration, take Listing 5 as the debuggee program, Listing 11 as the breakpoint declarations. Figure 2 shows a complete evaluation process of the breakpoint closeRead_if. Column *Code* lists code where the primitive breakpoints hit. Columns *Hit History* and *Values* describe the runtime states of the lists *hittingHistory* and *values* respectively. Column *Evaluation* has two sub-columns which represent the two-stage evaluation respectively. Sub-column *path* represents matches on the execution path and sub-column *If expr.* represents the test of the **if**() expression. "T" and "F" stand for true and false. When a path is found, a "T" is put in the sub-column *path*. A list representing the indexes of the hit history is put after "T", as in like T{0,1}.

Code	Hit History		Values		Evaluation	
	Index	Bp. ref.	Index	Map	path	If expr.
s1.close( )	0	close	0	"cStream" →s1	F	-
s2.read( )	1	read	1	"rStream" →s2	T{0,1}	F
s2.close( )	2	close	2	"cStream" →s2	F	-
s1.read( )	3	read	3	"rStream" →s1	T{2,3}	F
					T{0,3}	T

Figure 2: A table showing how a composite breakpoint stores the hit history and the bound values

The breakpoints are hit sequentially from top to bottom in the table and the evaluation is performed accordingly. When the program reaches s2.read(), a path is found with indexes {0,1}. Then, the composite breakpoint uses the indexes of the path to retrieve values, which are required by the **if**() condition, from the *values* list. However, the condition "cStream==rStream" does not hold. When the program reaches s1.read(), the composite breakpoint finds a path with indexes {2,3} but the **if**() condition again does not hold. Then, another path with indexes {0,3} is found and satisfies the **if**() condition. The composite breakpoint closeRead_if is hit and it produces a suspension message.

4.3 Named Advices

Bodden et al. [5], as well as Marot and Wuyts [16] proposed named advices for the purpose of uniquely identifying an advice. They extended the AspectJ syntax to achieve this goal. We name advices for referring to them in pointcut **adviceexecution**(). Besides, we do not intend to extend the syntax of the debuggee program, because the effort integrating it with the rest of our debugging infrastructure [19] is not trivial. Annotations are not supported in the *abc* compiler, which is part of our tool chain. Therefore, we choose to use comments as the approach for naming advices.

Listing 12 shows a **before** advice with a comment naming the advice as firstBefore. The programmer can refer to this advice in the breakpoint declaration. For example, **adviceexecution**(Azpect.firstBefore). During compilation, methods with unique identifiers as method names are created. For simplicity, we do not use the specified name as method name, but let the compiler decide the name as usual. Instead, we keep a mapping between the advices' *virtual names* as specified in the comment and their *compiled name*, as Listing 13 shows. During compilation, a validator checks whether there are ambiguous virtual names and prints an error message, if so.

```
aspect Azpect() {
  /**
   * @advicename=firstBefore
   */
  before() : call(...) {}
}
```

Listing 12: A named advice

```
<map>
  <entry>
    <virtual>firstBefore</virtual>
    <compiled>before$1</compiled>
  </entry>
</map>
```

Listing 13: A naming map

When a breakpoint declaration using **adviceexecution**() is sent to NOIRIn, which reads the virtual advice name and replaces it with the compiled name by using the naming map. When this breakpoint is hit, a hitting message is produced. The creation of the message replaces the compiled name with the virtual name.

4.4 User Interface

We implemented a dedicated user interface to manage and set breakpoints. The snapshots are given in Figure 3. The breakpoint view (Figure 3(a)) lists all the breakpoints and it has five columns, which are for (de)activation, presenting the breakpoint names, giving the complete declarations, showing whether a breakpoint is hit, and counting the hits respectively.

The panel shown in Figure 3(c) is for creating or editing a breakpoint. From top to bottom, it consists of four parts. The *message* part is for showing error or warning messages. The *script* part is for writing the breakpoint declaration. The *hit count* part is for specifying the expected hit count. The *graph* part is for presenting the reference relationships to other breakpoints. The panel in Figure 3(c) constructs a breakpoint which refers to breakpoints *close* and *read* in its expected path. The corresponding graph shows their hierarchical relationship. The declaration detail is shown when hovering the mouse over a label in the graph. Each label has a check box where programmers can (de)activate the corresponding breakpoint. If the declaration refers to breakpoint names which do not exist, as shown in Figure 3(d), an error message is shown on the *message* part and names in the graph are labeled with "invalid name".

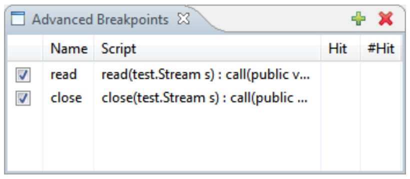
Moreover, we provide functionalities to ease the burden of typing static information, such as signatures and line locations. By using the context menu (Figure 3(b)) in the editor, text can be generated according to where the cursor is. For example, if the programmer selects a method name, the signature of the method can be generated. The generated text is appended to the *script* part of the breakpoint panel. Therefore, programmers do not have to manually type such information and can further customize the generated texts, such as replacing part of the text with wildcards.

4.5 Runtime Interactivity

We allow programmers to add, delete, and update breakpoints at editing time and during the execution of the debuggee program. The addition, deletion, or update of a breakpoint does not only changes itself but may also propagate the effect to other breakpoints. A breakpoint is invalid if its declaration contains invalid reference names. When a composite breakpoint becomes invalid, it does not make sense to keep its recorded history. Therefore, the runtime changes may alter the behavior of breakpoints. Other operations, such as additions and deactivations, do not affect the validity of other breakpoints. In the following list, we discuss these effects in detail.

- If a breakpoint is deleted, all breakpoints directly or indirectly referring to it become invalid. An invalid breakpoint discards the information it has recorded. This effect propagates until no more valid breakpoint can become invalid.

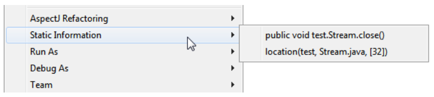
- The addition of a breakpoint triggers a re-compilation of all invalid breakpoints. If the added breakpoint



Advanced Breakpoints

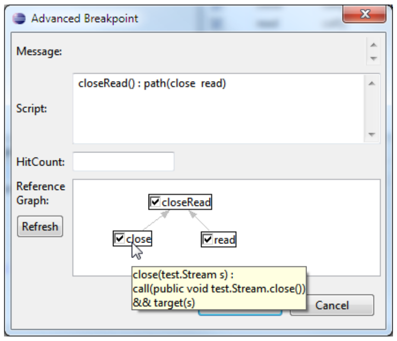
	Name	Script	Hit	#Hit
✓	read	read(test.Stream s) : call(public v...		
✓	close	close(test.Stream s) : call(public ...		

(a) The breakpoint view



public void test.Stream.close()
location(test, Stream.java, [32])

(b) The context menu



Message:

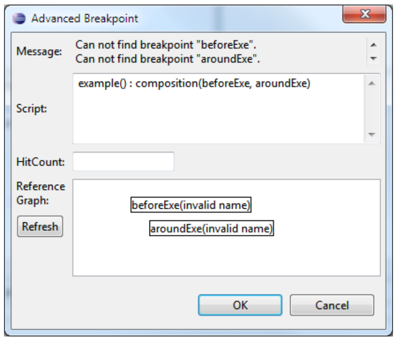
Script: closeRead() : path(close read)

HitCount:

Reference Graph: closeRead, close, read

close(test.Stream s) :
call(public void test.Stream.close())
&& target(s)

(c) The breakpoint panel



Message: Can not find breakpoint "beforeExe".
Can not find breakpoint "aroundExe".

Script: example() : composition(beforeExe, aroundExe)

HitCount:

Reference Graph: beforeExe(invalid name), aroundExe(invalid name)

(d) The panel with invalid names

Figure 3: Snapshots of the user interface

matches the missing part in previously invalid breakpoints, the invalid breakpoints become valid. This effect propagates until no more invalid breakpoint can become valid.

- If the script of a breakpoint is updated, this can be deemed as a deletion followed by an addition.

4.6 Performance

To get an indication of the runtime overhead imposed by our proposed debugging approach, we performed preliminary micro benchmarks. For this purpose, we use an application performing *bubble sort* and added a dummy aspect to be able to use our **adviceexecution** and **composition** pointcuts. We first executed this selected program on a plain Java Virtual Machine; second, we added breakpoints which use all the new features introduced in BPL and executed this on NOIRIn+BPL. The program uses fields instead of local variables for storing all temporary values, therefore the join points are dense in the program execution, and NOIRIn+BPL performs evaluation of breakpoints at each join point. This represents *the worst case* of using our approach. To avoid measuring the time spent during the suspensions, we change the infrastructural code to let breakpoints only print a message when they are hit. Though breakpoints do not cause any suspension throughout the whole execution, the evaluations whether each breakpoint is hit are actually performed. Comparing the execution times on the plain Java Virtual Machine and on NOIRIn+BPL with the breakpoints shows a slow-down of 20 times. For very short running debug sessions such a slow-down is already acceptable. Since we have focused on the semantics rather than on an efficient implementation, currently many breakpoint evaluations are redundant. In future work, we will also consider optimizations to avoid these redundant evaluations.

We have not yet analyzed the memory overhead which may especially be imposed by the **path**() pointcuts. Based on preliminary experiments, nevertheless, we do not expect this memory overhead to be limiting in typical, short-running debugging sessions.

5. EXAMPLES

5.1 Debugging Action Compositions

In this example, we add a new requirement to the **Scenario 4**, which is about recording execution history. A closed stream can be re-opened before it reads data in the context of the vital parts of the program. The **around** advice (lines 3–11) in Listing 14 implements this new requirement. Lines 13–17 contain a bug, which is reading data from a closed stream. The ellipses mean that statements may not be close to each other. We use the italic font for the variables *stream* to indicate that they refer to the same object but may use different names. The symptom of the bug is observed at line 17 but the root cause is the premature close() at line 15. The purpose of debugging is to intentionally suspend the program at the root cause. In the following paragraphs, we describe how a typical debugging process is performed in different solutions.

```
 1 aspect SafeStream {
 2   pointcut VitalPart() : ...;
 3   Object around(Stream s) :
 4       call(* Stream.read()) && target(s) &&
 5       cflow(VitalPart()) {
 6     if(s.isClosed()) {
 7       s.open();
 8     }
 9     Object result = proceed();
10     return result;
11   }
12 }
13 Stream stream;
14 ...
15 stream.close();   // the root cause
16 ...
17 stream.read();  // an exception is thrown.
```

Listing 14: An aspect with an advice that checks whether a Stream is closed and a program slice that performs operations on a Stream

The traditional debugging

Before starting the debugging, the programmer should consider where to put the breakpoints. Reading the exception message, she knows that the root cause must be an invocation of close(). A natural thought is setting a breakpoint to the call site of close(). However, there may be many invocations of close() in the program and setting breakpoints to each of them is troublesome. Therefore, putting the breakpoint to the body of close() and then tracing back to its call sites is more feasible.

1. Set two breakpoints to the body of methods close() and read(). Start debugging.
2. When the program is suspended, note the hit count of the breakpoint and the object identity of the Stream object. Resume debugging.
3. Repeat step 2 until the exception occurs. The first debugging session terminates.
4. Check the note and find out where the problematic Stream was closed by comparing the object identity. Record the corresponding hit count of the breakpoint set in close().
5. Delete or deactivate the breakpoint set in read(). Use the recorded hit count to restrain the breakpoint set in close(). Start debugging again.
6. When the program is suspended, it is in the execution of close() invoked by the root cause. The source, which is located at the second top stack frame in the stack trace, is the root cause.

Besides finding appropriate places to set breakpoints, this process spends significant effort on manually noting record and searching history.

The program solution

In this solution, the programmer realizes that using the program in Listing 6 will result in many false positives. The **around** advice interrupts the matching of the trace pattern "close read" because it calls Stream.open(). We see two ways of handling this. First, Stream.open() can be considered in the pattern. Second, and more straightforward, calls to Stream.read() which are advised by the **around** advice can be excluded. Suppose the programmer selects the latter way, the desired condition should contain the negation of the pointcut **cflow**(VitalPart()) (Listing 14, line 5).

1. Add the following aspect and tracematch to the program. Replace the ellipsis on line 11 with the definition of the pointcut VitalPart() (Listing 14, line 2). Set a breakpoint at line 13. Start debugging.

```
 1 Aspect debugging {
 2   private int hitcount=0;
 3   before() : execution(* Stream.close()) {
 4     hitcount++;
 5     print(hitcount);
 6   }
 7 }
 8 tracematch(Stream s) {
 9   sym close before : call(* Stream.close()) && this(s);
10   sym read before : call(* Stream.read()) && this(s)
11     && !(cflow(...));
12   close read {
13     // set a breakpoint on this line
14   }
15 }
```

2. When the program is suspended, read the printed hit count produced by line 5. The first debugging session terminates.

3. Delete or deactivate the set breakpoint. Use the recorded hit count to set a breakpoint in the body of Stream.close(). Start debugging again.
4. When the program is suspended, it is in the execution of close() invoked by the root cause. The method execution, which is located at the second top stack frame in the stack trace, is the root cause.

This process automates the manual work in the previous process. The most effort spent concentrates on writing the program, especially the condition that excludes the **around** advice. Designing such a correct condition may be non-trivial. For example, a condition requires to exclude or include multiple advices.

The BPL solution

The programmer has the same flow of thought as she does in the program solution. She needs to exclude the calls to Stream.read() where the **around** advice is applied.

1. Name the **around** advice as "aroundAdvice" and define the following five breakpoints. Line 6 shows how the **around** advice is excluded. Method signature such as Stream.close() can be generated by using the user interface. Activate only close_readNoAround. Start debugging.

```
 1 close(Stream s) : call(* Stream.close()) && target(s);
 2 read(Stream s) : call(* Stream.read()) && target(s);
 3 aroundAdvice() :
 4     adviceexecution(SafeConnection.aroundAdvice);
 5 read_NoAround(Stream s) :
 6     composition(read, !aroundAdvice) &&
 7     bind(read(s));
 8 close_readNoAround(Stream s) :
 9     path(close read_NoAround) &&
10     bind(read_NoAround(s), close(s));
```

2. When the program is suspended, close_readNoAround prints the following information on the console. The first debugging session terminates.

```
**************** close_readNoAround ****************
matched path (close read_NoAround)
close examples\StreamTest.java(line 10, hitcount 3)
read_NoAround examples\MyLogger.java(line 30, hitcount 1)
−−read examples\MyLogger.java(line 30, hitcount 2)
```

3. Delete or deactivate close_readNoAround. According to the printed information, activate the breakpoint close and configure the hit count with 3. Start debugging again.
4. When the program is suspended, the root cause is found.

This process overcomes the shortcomings of the previous two processes. It not only automates manual works but also constructs the condition in a straightforward way. The **composition**() expression is an intuitive way for expressing a certain action composition and it does not require a sophisticated analysis.

5.2 Debugging Dereference Operations

JabRef is an open source bibliography reference manager. We have scanned the commits in its subversion repository to find all the revisions with reports containing the keyword "bug". Revision #25 reports a fixed null pointer bug. Listing 15 and 16 show the buggy program and the revised program. Line 4, which tests whether frame.basePanel() is

```
1 class SearchManager {
2   public void actionPerformed(ActionEvent e) {
3     if (e.getSource() == escape)
4
5       frame.basePanel().stopShowingSearchResults();
6     ...
7   }
8 }
```
Listing 15: A buggy program

```
1 class SearchManager {
2   public void actionPerformed(ActionEvent e) {
3     if (e.getSource() == escape)
4       if (frame.basePanel() != null)
5         frame.basePanel().stopShowingSearchResults();
6     ...
7   }
8 }
```
Listing 16: A revised program

null, is added in the revised program. By reverse engineering, we can deduce that a *NullPointerException* is thrown in the execution of line 5 of the buggy program. There are two possible root causes, one is the field frame and another is the expression frame.basePanel(). The exception occurs on the line where frame is accessed the first time in method actionPerformed. Therefore, the possibility that frame stores a null value cannot be excluded. The programmer needs to check both dereference operations during debugging.

The traditional debugging

1. Set a breakpoint on line 5. Start debugging.
2. When the program is suspended, inspect the value of the field frame.
3. The frame is not null. Deduce that the expression frame.basePanel() returns a null value.
4. The root cause is found. Terminate debugging.

In this process, most effort concentrates on finding the expression that returns null. This effort increases with the number of dereference operations on the same line. The programmer has to inspect each dereference operation until she can decide which one is the root cause. There are usually two ways of inspection. One is to copy and evaluate an expression. Another is to repeatedly perform "step into" and "step return" and meanwhile note which dereference operation the debugger comes to.

Another solution is putting each problematic dereference operation in a separate line and rerun the program. Then, the line number from the error message indicates that the dereference operation on that line is the cause. However, the chopped format of code is not as readable as it was. After the bug is fixed, the code fragments need to be put back together. Similar to the other traditional solution, it does not scale well when the number of dereference operations increases. Besides, this option requires changing the source code, which is not generally possible.

The program solution

1. Manually code the following aspect, add it to the debuggee project, and set a breakpoint at line 9 in the added program. Start debugging.

```
1  public aspect ProgramSolutionAspect {
2    before(Object receiver) :
3        (call(* *.*(..)) || get(* *.*))
4        && target(receiver) && withincode(public void
5          SearchManager.actionPerformed(ActionEvent))
6        && if(receiver == null) {
7      int line = thisJoinPoint.getSourceLocation().getLine();
8      if(line == 5) {
9        // set a breakpoint on this line
10     }
11   }
12 }
```

2. When the program is suspended, inspect the variable **thisJoinPoint** and find out the signature of the method call or the field access.
3. The signature contains stopShowingSearchResults(). Deduce that the expression frame.basePanel() returns a null value.
4. The root cause is found. Terminate debugging.

This process automates the task of finding the root cause but complicates the way of setting breakpoints. The program describes the task and constructs a place for setting the breakpoint. It needs non-trivial designing and implementation effort.

The BPL solution

1. Set a breakpoint with the following breakpoint declaration using the user interface to generate the **location**() expression. Start debugging.

```
bp() : checkNPE() && location(
        "net\sf\jabref", "SearchManager.java", [5]);
```

2. When the program is suspended, the breakpoint prints the following information on the console. Deduce that the expression frame.basePanel() returns a null value.

```
**************** bp ****************
Panel.stopShowingSearchResults() has a null receiver.
```

3. The root cause is found. Terminate debugging.

This process combines the advantages of the previous two processes. It not only automatically detects the root cause, but also requires only trivial effort for setting the breakpoint. This comparison highlights the great convenience and efficiency of using BPL.

6. RELATED WORK

We categorize the related work into three groups, which are breakpoints, debuggers, and pointcut languages.

6.1 Breakpoints

Modern IDEs have developed some advanced breakpoints. The IntelliJ Java debugger supports temporal dependency between two breakpoints: If a breakpoint A depends on another breakpoint B, then A cannot be hit until B is hit. Visual Studio allows setting breakpoints on a specific call to a function by using the stack trace. Such a breakpoint suspends the program when the call stack is exactly the same as the one it was set on. Nevertheless, these advanced breakpoints are developed based on line breakpoints, which hardly show why breakpoints are placed there. BPL uses programs to specify breakpoints and the intention of using the breakpoints, like suspending the program at method calls or field accesses, are explicit.

Chern and De Volder [6] proposed the control-flow breakpoint to suspend the program according to the state of the stack trace. The control-flow breakpoint can specify that an event should or should not occur in the control-flow of another event. The breakpoint specification can be gradually refined at runtime until only the expected suspensions occur. In our approach, control flow refinements result in a breakpoint declaration which consists of multiple *cflow* expressions. In addition to the control flow, we also support sequential execution pattern.

The stateful breakpoint [4] allows programmers to suspend the program when some line breakpoints are hit in an expected order and certain values at those hits are coincident. A stateful breakpoint consists of three parts: a set of named line breakpoints, variables bound by the line breakpoints, and an execution trace composed by the names of the line breakpoints. Our composite breakpoint has two main differences to the stateful breakpoint. First, we provide more flexible ways specifying conditions on the bound variables by explicitly using if(). Second, a primitive breakpoint in BPL, once it has been defined, can be referenced by multiple composite ones. A primitive breakpoint is not reusable in stateful breakpoints.

6.2 Debuggers

Bugdel [18] is an AO debugging system in which programmers can set AO breakpoints by using dedicated graphical user interfaces. It can insert statements at a breakpoint to specify what to do when the breakpoint is hit. Its breakpoint model is join-point-shadow based, which is more fine-grained than line breakpoints. However, breakpoints defined in Bugdel are independent from each other. Thus, they cannot be used to compose higher-level breakpoints.

JavaDD [10] is a declarative debugger on which programmers can perform queries over the recorded execution history. Like other query-based debuggers, JavaDD records all the salient events, such as method calls, or field assignments, at runtime. In our approach, the breakpoint declarations are similar to queries but they are written before and applied to the following execution. Our approach records only the interesting values and, thus, programmers cannot query values which were not recorded.

Ducassé [7] complained that line-based breakpoints do not have semantics and, therefore, proposed Coca, which is a debugger using only events related to source abstractions as queries. Our approach does not throw away line-based breakpoints, because they are sometimes easier to be specified and more straightforward than breakpoints using source abstractions. Besides, Coca uses Prolog, which is completely different from the language of the debuggee programs, as the query language. Learning Prolog increases the cost of using debugging facilities. Our approach aims at minimizing the learning effort by using a pointcut language using abstractions that are natural to programmers of object-oriented and aspect-oriented programs, namely one similar to AspectJ.

6.3 Pointcut Languages

Tracematch [1] uses regular expressions consisting of references to primitive pointcuts to specify an expected execution trace. It supports free variables in specifying a trace. Therefore, a matched trace depends on not only the order of events but also the associated variables of the events. The design of the composite breakpoint in BPL is greatly inspired by Tracematch. However, there are two fundamental differences. First, a *tracematch* is a standalone unit and other *tracematch*es cannot reuse its members including definitions of primitive events. In BPL, a primitive breakpoint can be referred by any number of other breakpoints. Second, there are several event kinds, such as *before* and *after returning*, in Tracematch. BPL is only interested in *before* because we want to suspend the program before the interesting events occur.

Oarta [16] extends AspectJ with features, which are similar to some of BPL. It supports *named advices* by putting a name in the declaration of an advice. It also allows declaring precedence at the advice level. Our approach does not change the syntax of AspectJ and it names advices by using Java comments. Besides, our composition specification targets finding advice compositions at runtime instead of defining precedence rules for weaving.

7. CONCLUSION AND FUTURE WORK

In this paper, we identified five scenarios of using breakpoints. These scenarios are frequently encountered but not supported sufficiently by existing breakpoints. Programmers need to manually perform some repetitive tasks, thus debugging efficiency is decreased.

Targeting all scenarios, we proposed a breakpoint language (BPL) which models the breakpoint as a first-class values. Breakpoints are named and they are defined by AspectJ-like pointcuts which use comprehensible source-level abstractions. We devised five completely new pointcut designators and improved two or AspectJ's pointcut designators. In our language, primitive and composite breakpoints are treated uniformly and the composition level can be infinite. It is the first language to support selecting join points with a specific advice composition.

We illustrate the usage of our approach by means of two example walkthroughs. The examples show that BPL has the following advantages over the traditional debugging and the approach using other languages.

- It allows programmers to describe the logic relationship between multiple breakpoints with succinct code.
- It allows using pointcut **composition**() to express an advice composition in a straightforward way.
- It automatically records and prints information, such as the source location and the hit count, of a hit breakpoint and its referenced breakpoints. The information is helpful for localizing the root cause in an additional debugging sessions.

The pointcut **composition**() can also be used for other purposes. For example, it can verify whether a certain advice composition exists or not in the program. Another example is handling the fragile pointcut problem. Sometimes, changes to a pointcut may unexpectedly exclude or include some join points. Therefore, behavior occurring at these join points becomes undesired. To find the join-point differences, the following breakpoint declarations can be used.

```
oldOne() : adviceexecution(someAzpect.oldAdvice);
newOne() : adviceexecution(someAzpect.newAdvice);
inOldNotInNew() : composition(oldOne, !newOne);
inNewNotInOld() : composition(newOne, !oldOne);
```

Breakpoint inOldNotInNew is hit on the excluded join points and breakpoint inNewNotInOld is hit on the newly included ones. In this way, the program execution is suspended at the join points which are (not) advised in both the old and the new versions of the program. The two breakpoints narrow down the potential places causing the undesired behavior.

The BPL is motivated by some ad hoc scenarios based on our past experiences and observations. To standardize their usage, a systematic requirement analysis is needed to enhance BPL's generality. Some language features can be explored more in the future. For example, the **path**() expression should support more types of path patterns, the runtime interactivity should be fault-tolerant in case of an accidental update, etc. Also, we plan to build an omniscient debugger for advanced-dispatching languages and our BPL can be reused as part of the built-in queries.

8. ACKNOWLEDGEMENTS

This work is partly funded by the Chinese Scholarship Council (CSC Scholarship No.2008613009).

9. REFERENCES

[1] C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding trace matching with free variables to AspectJ. *SIGPLAN Not.*, 40(10):345–364, Oct. 2005.

[2] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. abc: an extensible AspectJ compiler. In *Proceedings of the 4th AOSD*, pages 87–98, New York, NY, USA, 2005. ACM.

[3] C. Bockisch, A. Sewe, H. Yin, M. Mezini, and M. Aksit. An in-depth look at alia4j. *Journal of Object Technology*, 11(1):7:1–28, Apr. 2012.

[4] E. Bodden. Stateful breakpoints: a practical approach to defining parameterized runtime monitors. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ESEC/FSE '11, pages 492–495, New York, NY, USA, 2011. ACM.

[5] E. Bodden, F. Chen, and G. Rosu. Dependent advice: a general approach to optimizing history-based aspects. In *Proceedings of the 8th ACM international conference on Aspect-oriented software development*, AOSD '09, New York, NY, USA, 2009. ACM.

[6] R. Chern and K. De Volder. Debugging with control-flow breakpoints. In *Proceedings of the 6th international conference on Aspect-oriented software development*, AOSD '07, pages 96–106, New York, NY, USA, 2007. ACM.

[7] M. Ducassé. Coca: an automated debugger for C. In *Proceedings of the 21st international conference on Software engineering*, ICSE '99, pages 504–513, New York, NY, USA, 1999. ACM.

[8] P. E. A. Dürr. *Resource-based verification for robust composition of aspects.* PhD thesis, Enschede, June 2008.

[9] M. Eisenstadt. My hairiest bug war stories. *Commun. ACM*, 40(4):30–37, Apr. 1997.

[10] H. Z. Girgis and B. Jayaraman. JavaDD: a declarative debugger for java. Technical report, 2006.

[11] B. Hailpern and P. Santhanam. Software debugging, testing, and verification. *IBM Syst. J.*, Jan. 2002.

[12] A. Hannousse, R. Douence, and G. Ardourel. Static analysis of aspect interaction and composition in component models. In *Proceedings of the 10th ACM international conference on Generative programming and component engineering*, GPCE '11, pages 43–52, New York, NY, USA, 2011. ACM.

[13] E. Katz and S. Katz. Incremental analysis of interference among aspects. In *Proceedings of the 7th workshop on Foundations of aspect-oriented languages*, FOAL '08, New York, NY, USA, 2008. ACM.

[14] B. P. Lientz and E. B. Swanson. *Software Maintenance Management.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1980.

[15] A. Marot and R. Wuyts. Detecting unanticipated aspect interferences at runtime with compositional intentions. In *Proceedings of the Workshop on AOP and Meta-Data for Software Evolution*, RAM-SE '09, New York, NY, USA, 2009. ACM.

[16] A. Marot and R. Wuyts. Composing aspects with aspects. In *Proceedings of the 9th International Conference on Aspect-Oriented Software Development*, AOSD '10, New York, NY, USA, 2010. ACM.

[17] I. Nagy. *On the design of aspect-oriented composition models for software evolution.* PhD thesis, Enschede, June 2006.

[18] Y. Usui and S. Chiba. Bugdel: An aspect-oriented debugging system. In *Proceedings of the 12th Asia-Pacific Software Engineering Conference*, APSEC '05, pages 790–795, Washington, DC, USA, 2005. IEEE Computer Society.

[19] H. Yin, C. Bockisch, and M. Aksit. A fine-grained debugger for aspect-oriented programming. In *Proceedings of the 11th annual international conference on Aspect-oriented Software Development*, AOSD '12, New York, NY, USA, 2012. ACM.