Debugging with Control-flow Breakpoints

Rick Chern and Kris De Volder
Department of Computer Science
University of British Columbia
2366 Main Mall
Vancouver BC Canada V6T 1Z4
rchern,kdvolder@cs.ubc.ca

Abstract

Modern source-level debuggers support dynamic breakpoints that are guarded by conditions based on program state. Such breakpoints address situations where a static breakpoint is not sufficiently precise to characterise a point of interest in program execution. However, we believe that current IDE support for dynamic breakpoints are cumbersome to use. Firstly, guard conditions formulated in (non-aspect-oriented) source-languages cannot directly express control-flow conditions, forcing developers to seek alternative formulations. Secondly, guard-conditions can be complex expressions and manually typing them is cumbersome.

We present the Control-flow Breakpoint Debugger (*CBD*). *CBD* uses a dynamic pointcut language to characterise control-flow breakpoints—dynamic breakpoints which are conditional on the control-flow through which they were reached. *CBD* provides a "point-and-click" GUI to specify and incrementally refine control-flow breakpoints, thereby avoiding the burden of manually editing the potentially complex expressions that define them.

We performed 20 case studies debugging and fixing documented bugs in 3 existing applications. Our results show that dynamic breakpoints in general are useful in practice, and that *CBD*'s GUI allows specifying them adequately in the majority of cases.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids*

General Terms

Experimentation, Languages

Keywords

Pointcut language, Debugger interface, Case studies

AOSD 07, March 12-16, 2007, Vancouver, Canada


```
public class ActionSaveProject {
  public void actionPerformed(ActionEvent e) {
    ...
    ProjectBrowser.getInstance().trySaveAs(...); // line B
    ...
  }
}

public class ActionSaveProjectAs {
  public void actionPerformed(ActionEvent e) {
    ...
    ProjectBrowser.getInstance().trySaveAs(...);
    ...
  }
}

public class ProjectBrowser {
  public void trySaveAs(...) {
    ...
    // line A
    ...
  }
}
```

Figure 1: Motivating example.

1. Introduction

Modern IDEs offer sophisticated source-level debugger GUIs as part of their arsenal of tools. In this paper we are concerned with their mechanism for setting breakpoints. We distinguish two kinds of breakpoints, static breakpoints and dynamic breakpoints. A static breakpoint corresponds one-to-one with a specific line of code. A dynamic breakpoint on the other hand depends on an additional runtime condition.

The typical source-level debugger provides a straightforward point-and-click UI to create a static breakpoint. Static breakpoints are however not always sufficiently precise because they may be reachable from execution contexts the developer is not interested in. We believe that such situations occur reasonably often in practice and that state-of-the-practice debuggers do not provide adequate support for them.

1.1 Motivating Example

We illustrate the problem with a motivating example. The example was simplified for presentation purposes but is based on a real debugging scenario lifted from case 19 of our case studies.

In this session the developer was fixing a bug in

the ArgoUML GUI. A skeleton of some relevant code is shown in Figure 1. At some point in the session the developer wanted to put a breakpoint at *line A*. As the developer was trying to reproduce the bug, this line was reached from two different calling contexts. One was in response to handling the selection of "Save As..." from the "File" menu in the GUI, through the `ActionSaveProjectAs.actionPerformed()` method. The second one was in response to handling the selection of "Save" from the "File" menu through `ActionSaveProject.actionPerformed()`. Deliberately triggering the bug required selecting both "Save As..." and "Save" in sequence, but the execution of *line A* was only of interest when in response to the selection of "Save". In trying to fix the bug the developer repeatedly made some changes and performed the GUI actions that triggered the bug.

We will assume the standard Eclipse JDT debugger[11] is used in the rest of this discussion, but other modern IDEs provide similar functionality. Using the standard Eclipse JDT debugger, there are several options to try to obtain the desired breaks in execution.

The first option is to simply set a static breakpoint at *line A*. This option is unsatisfactory because program execution suspends every single time *line A* is executed, and one in two executions is from an uninteresting dynamic calling context.

A second option is to set static breakpoints at the interesting call site (*line B*) instead of at *line A*. Unfortunately, the ability to precisely mark the point of interest is then lost. This is problematic if this point is located in a large and complicated method body.

A third option is to use two static breakpoints in conjunction. For example, the *line A* breakpoint could be temporarily deactivated. Then a breakpoint could be set at *line B*. Upon reaching the *line B* breakpoint the programmer could then reactivate the breakpoint at *line A*. This workaround is used frequently by one of the authors and provided the initial inspiration for the work in this paper. It is unsatisfactory because repeatedly activating and deactivating the breakpoint is cumbersome and forces the programmer to suspend the program at *line B* even though he is not directly interested in it.

A fourth option is to attach a dynamic guard expression to the breakpoint at *line A*. The JDT debugger allows a programmer to attach an arbitrary Java expression as a guard to a breakpoint. In this case the following expression could be used:

```
EventQueue.getCurrentEvent().toString().endsWith(
                                    "text=Save Project]")
```

Since the guard condition must be expressed in Java and must be based on program state as captured by program variables, the desired control-flow condition cannot be directly expressed. Instead, the above expression relies on particular objects that happen to exist and happen to contain recognisable patterns in their debugging strings. In general, finding an appropriate expression is not obvious because different control-flow histories do not necessarily imply that the corresponding program states are easily distinguished. In the worst case, auxiliary code may need to be added to the program for the sole purpose of recording program history and expressing a breakpoint condition.

Options one, two, and three in this motivating example illustrate how static breakpoints can be insufficient in some debugging scenarios. The fourth option illustrates that state-of-the-practice debugging tools do not adequately support dynamic breakpoints. Specifically, it exemplifies how dynamic conditions based on control-flow instead of program state cannot be directly expressed with modern debuggers, and that manually formulating guard conditions can be difficult.

1.2 Approach

We now outline our approach to tackling the problem described above. To facilitate convenient discussion, from here on we will use the term "control-flow breakpoints" to refer to dynamic breakpoints that are characterised in terms of control-flow history (i.e. how the breakpoint was reached) rather than in terms of the state of program variables.

A central hypothesis of our work is that for practical debugging scenarios control-flow breakpoints are often more adequate in cases where guards based on variable state are unnatural. This hypothesis is at odds with state-of-the-practice debugging tools which implicitly force guards to be formulated only in terms of variable state.

A second belief inspiring our approach is that—even when offered a more appropriate mechanism that supports control-flow conditions—it is cumbersome for developers to *manually* enter potentially complex expressions.

To address both of these concerns, we propose a debugger that provides a "point-and-click" GUI to define control-flow breakpoints. This debugger GUI allows the user to specify control-flow breakpoints and incrementally refine them based on information from either the current debugging context or from recorded history. We call this debugger the Control-flow Breakpoint Debugger (*CBD*).

Supporting the debugger GUI is a dynamic pointcut language inspired by AspectJ [4] and related work on trace-based pointcuts (see Section 4). Our pointcut language characterises breakpoints as joinpoints (points in the execution of the program) where the program should suspend. It allows for selecting joinpoints based on temporal relations such as `cflow` (familiar from AspectJ), `before`, and `after`. We call our pointcut language Break Pointcut Language (*BPL*).

It should be stressed that although *BPL* is a central concept in our design, pointcut expressions serve mainly as a convenient internal representation of breakpoints. As such, the *CBD* GUI completely avoids the need for developers to manually enter pointcut expressions, and in fact our prototype does not even have a parser for *BPL* expressions.

1.3 Evaluation

To evaluate the practical expressiveness of control-flow breakpoints and *CBD*'s graphical user interface, we performed 20 case studies debugging and fixing documented bugs in 3 existing Java applications. The results confirm our three main claims:

1. Situations in which the additional precision of some kind of dynamic breakpoint condition could be useful occur in practice.

2. Control-flow breakpoints are adequate in the majority of these situations.

3. The purely "point-and-click" GUI of *CBD* supports

specifying control-flow breakpoints for a majority of practical debugging scenarios.

Our first claim establishes our hypothesis that there are situations in which dynamic breakpoint support is at least potentially useful. Having established that, we secondly claim that control-flow breakpoints are *expressive* enough to characterise the desired dynamic conditions in most of such situations. We believe that these two claims are novel because although numerous proposals of mechanisms to support dynamic breakpoints exist [8, 10], to the best of our knowledge no empirical data about the potential usefulness of such mechanisms in realistic debugging scenarios have been presented in the literature.

While the *CBD* GUI only supports specification of a subset of the full range of control-flow breakpoints, our third claim asserts that the *CBD* GUI is *expressive* enough to support creation of the majority of control-flow breakpoints required in our second claim.

1.4 Outline

The remainder of this paper is organised as follows. Section 2 describes Control-flow Breakpoint Debugger, Section 3 describes the process for evaluating *CBD* and the evaluation results, Section 4 provides an overview of previous related work, and Section 5 concludes this paper. Possible future work is outlined in Section 6.

2. The Control-flow Breakpoint Debugger

This section describes the Control-flow Breakpoint Debugger. *CBD* is implemented as a modification to the standard Eclipse JDT debugger and can be downloaded from `http://www.cs.ubc.ca/labs/spl/projects/cbd.html`.

We start with a simple use case, revisiting the motivating example. This illustrates how *CBD* addresses the issues described in Section 1 and also allows us to outline the general ideas behind *CBD*'s design. The remaining subsections provide a more detailed explanation of *CBD*'s features and the underlying pointcut language which plays a central role in its design and implementation.

2.1 Motivating Example Revisited

Reconsider the situation described in Section 1.1 and the example Java code shown in Figure 1. Recall that *line A* is reached multiple times during the debugging session in uninteresting contexts. We will now see how a developer might use *CBD* in this situation.

The programmer begins by setting a breakpoint at *line A*. Initially this breakpoint is purely static and behaves identically to a line breakpoint in a standard debugger.

CBD represents breakpoints by means of pointcut expressions in Break Pointcut Language, a pointcut language that is loosely based on AspectJ but modified for the specific purpose of representing breakpoints. For example, the static *line A* breakpoint corresponds to the pointcut expression:

$$\texttt{line\_execution}(l_A)$$

where l_A is a line signature, i.e. a string (containing a file and line number) uniquely identifying *line A*. After placing the *line A* breakpoint, the programmer reproduces the bug multiple times to examine program state during execution, occasionally making changes to the code in an attempt to fix the bug. When *line A* is reached for the first time in the

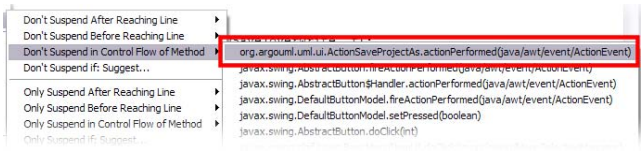

Figure 2: Selecting the first item on the stack for control-flow exclusion.

control-flow of `ActionSaveProjectAs.actionPerformed()` she realises that this dynamic context is uninteresting and would like to avoid it in the future. In terms of the breakpoint's pointcut, this can be accomplished by conjoining an additional condition to it.

The *CBD* GUI avoids manual entry of this condition by providing several menu options. Each option corresponds to a specific condition that is to be conjoined with the breakpoint. *CBD* derives the options either from the current execution context, or from (information recorded from) an execution context under which the breakpoint was previously triggered. Both of these two categories of options are intended to support a use case similar to the one from our example, where the programmer notices that a particular breakpoint is triggered in a dynamic context she is not interested in and wants to avoid this context in the future. A condition c_c that is true in the current uninteresting context can be used to avoid similar contexts in the future by conjoining the breakpoint pointcut with a negated version of c_c like so:

$$p' ::= p \texttt{ \&\& !} c_c$$

Alternatively a condition c_r recorded in a previous activation can be used to restrict to contexts that are similar to it like so:

$$p' ::= p \texttt{ \&\& } c_r$$

Thus the current uninteresting context can be eliminated from future executions if it is dissimilar (some property c_r does not hold) from a previously reached interesting context.

Two separate sets of menus labelled "Don't Suspend ..." and "Only Suspend ..." respectively correspond to restricting the pointcut negatively (based on the current activation) or positively (based on a previous activation). Several kinds of dynamic conditions are supported as different submenus, but the `cflow` condition was predominantly used in our actual case studies.

Returning to our example, to exclude the current activation in the context of `ActionSaveProjectAs.actionPerformed()` the developer chooses the "Don't Suspend in Control-Flow of Method" menu item. This pops up a submenu displaying the method signatures on the current call stack, as shown in Figure 2. She chooses the topmost item. This results in *CBD* redefining the *line A* breakpoint to the following pointcut:

$$\texttt{line\_execution}(l_A) \texttt{ \&\& !cflow(method\_execution}(m)\texttt{)}$$

where m is a method signature corresponding to `ActionSaveProjectAs.actionPerformed()`. This achieves the desired effect of avoiding future executions of this breakpoint in uninteresting contexts. Note that a similar effect could be achieved also by restricting the pointcut to

the control-flow of `ActionSaveProject.actionPerformed()` based on a previous activation.

2.2 The CBD Pointcut Language

A central element of *CBD*'s design and implementation is Break Pointcut Language. An overview of *BPL* is shown in Table 1. *BPL* expressions are convenient breakpoint representations that support both static and dynamic control-flow breakpoints. *BPL* is loosely inspired by AspectJ and resembles AspectJ's pointcut language in terms of its syntax and semantics.

Like AspectJ pointcuts, *BPL* expressions are composed out of atomic pointcuts which are combined through different pointcut combinators. These combinators include `&&`, `||`, `!`, and `cflow`. These "standard" combinators have the same meaning as they have in AspectJ.

The set of atomic pointcuts from which *BPL* expressions are composed differ slightly from those found in AspectJ, as the language is intended to support a debugger UI rather than an aspect-oriented programming language. *BPL* provides only four atomic pointcut types. The `method_execution` pointcut corresponds to AspectJ's `execution` pointcut and identifies a point in execution where a method-body is executed. However, contrary to AspectJ it does not support wild-card patterns in the method signature[1]. *BPL*'s `line_execution`, `method_entry`, and `method_exit` pointcuts have no counterparts in AspectJ. The `line_execution` pointcut serves the purpose of identifying the execution of a specific line of source code and is needed to support a typical debugger UI which allows setting breakpoints on specific lines of code. The `method_entry` and `method_exit` pointcuts are needed to support the setting of method entry and method exit breakpoints respectively in a typical debugger UI.

Besides the "standard" combinators, *BPL* has two additional pointcut combinators. The `before` and `after` pointcuts are similar to `cflow` in that they express a temporal relationship between the matched joinpoint and joinpoints matched by its pointcut parameter. The `after` combinator was added because we believed it could replace ad-hoc approaches which manually activate a breakpoint after another breakpoint has been reached, as was discussed in the motivating example. A `before` combinator was added for logical completeness.

Like AspectJ, *BPL* has a clear separation between static and dynamic pointcut expressions. This separation provides a natural mapping from a pointcut expression to combination of a static breakpoint and a dynamic condition in a debugger UI. Dynamic breakpoints in the debugger UI are represented using *BPL* pointcut expressions of the following forms:

$$\texttt{line\_execution}(l)\ \texttt{\&\&}\ p$$

$$\texttt{method\_entry}(m)\ \texttt{\&\&}\ p$$

$$\texttt{method\_exit}(m)\ \texttt{\&\&}\ p$$

where `line_execution`(l), `method_entry`(m), and `method_exit`(m) are static pointcuts that correspond respectively to the lines and methods in a typical debugger UI where breakpoints are created. The pointcut p is a residual dynamic pointcut which corresponds to the dynamic condition attached to a breakpoint in the debugger UI.

[1]Wild-card features were deemed to be unnecessary to support the *CBD* GUI. This does not mean that breakpoints specified in terms of wild-cards could not be potentially useful in some scenarios.

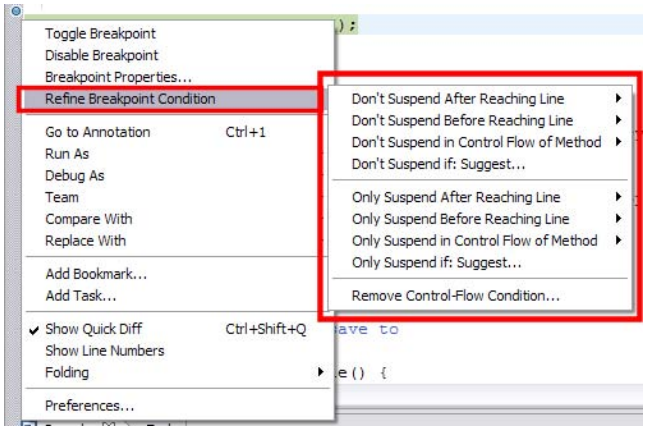

Figure 3: *CBD* menu structure.

2.3 The CBD User Interface

The Control-flow Breakpoint Debugger's graphical user interface is an extension of the Eclipse JDT debugger UI, which already provides a convenient interface for creating static breakpoints (i.e. breakpoints that correspond directly to *BPL*'s atomic pointcut expressions). *CBD* provides additional functionality to attach dynamic control-flow conditions to such breakpoints. This functionality is invoked through a context menu for a breakpoint marker as shown in Figure 3. The menu is labelled "Refine Breakpoint Condition" and contains three subsections.

Recall from Section 2.1 that *CBD* allows breakpoints to be refined based on control-flow information from either the current execution context ($p' := p$ `&&` `!`c_c) or a previously recorded execution context ($p' := p$ `&&` c_r). The "Don't Suspend ..." section is based on the current context and contains choices which allow the programmer to exclude joinpoints that have similar control-flow properties from being matched in the future. The "Only Suspend ..." section on the other hand is based on a recorded context and contains choices which allow restricting to joinpoints that have similar control-flow properties. In the current implementation only one context is recorded for each breakpoint: the first context in which the breakpoint is triggered. Both menu sections have very similar structure and perform very similar functionalities. The only real difference between them is that they are based on a different example context (current or recorded) and use the derived conditions either in a negated or non-negated form.

In each section, the first three menus follow the structure of *BPL* and provide three different options which correspond to the three temporal relationships—`cflow`, `before`, and `after`—supported by it. These options can be used by a developer who has a fairly precise idea of what kind of condition to use. The fourth and final item is intended to provide a more open list of suggestions which are deemed to be most likely to be useful, based on some simple heuristics. We now discuss each menu item in each menu section in more detail.

Syntax	Matched Joinpoints
Atomic Pointcuts	
`line_execution(`l`)`	All executions of the source-code line specified by l
`method_execution(`m`)`	All executions of the method body specified by the signature m
`method_entry(`m`)`	All entries to the method specified by the signature m
`method_exit(`m`)`	All exits from the method specified by the signature m
Temporal Combinators	
`cflow(`p`)`	All points in execution within the control-flow of the points matching the pointcut p
`after(`p`)`	All points in execution after and including the first point matching the pointcut p
`before(`p`)`	All points in execution before the first point matching the pointcut p
Logical Combinators	
`!`p	All points in execution except points matching the pointcut p
p_1 `&&` p_2	Points in execution matching both of the pointcuts p_1 and p_2
p_1 `||` p_2	Points in execution matching either of the pointcuts p_1 and p_2

Table 1: Summary of Break Pointcut Language

2.3.1 The "Don't Suspend . . . " Menu Section

The "Don't Suspend in Control-Flow of Method" submenu displays the current call stack. When a method signature m is selected from this menu, the pointcut is modified as follows:

$$p' := p \texttt{ \&\& !cflow(method\_execution(}m\texttt{))}$$

This will avoid suspending in any future context where this particular method is on the control stack.

The "Don't Suspend Before Reaching Line" submenu contains as options lines that already have breakpoints on them but that have not yet been reached in execution. When a line l is selected from the menu, the pointcut is modified as follows:

$$p' := p \texttt{ \&\& !before(line\_execution(}l\texttt{))}$$

This will avoid suspending at the current breakpoint until after the selected line is reached.

The "Don't Suspend After Reaching Line" submenu shows break lines that have breakpoints which have already been reached in execution. When a line l is selected the breakpoint pointcut is modified as follows:

$$p' := p \texttt{ \&\& !after(line\_execution(}l\texttt{))}$$

This will avoid suspending at the current breakpoint until the program is restarted, and will no longer suspend after l is reached.

The "Don't Suspend if: *Suggest*" menu item provides a single suggestion (see Figure 4). This suggestion is based on our experience that the `cflow` condition is the most frequently used condition in practice. It picks the most recent method on the stack that is in the current context but not in the recorded context (or simply the next most recent method on the stack if the breakpoint was not triggered before and hence has no recorded context). The current suggestion heuristics are provisional and will likely be replaced with more sophisticated algorithms in the future (see Section 6).

2.3.2 The "Only Suspend . . . " Menu Section

The structure and functionality of this menu section is similar to the "Don't Suspend . . . " menu. The difference is that the condition is applied to the selected breakpoint without negation and the suggested conditions are not extracted from the current context, but from a recorded context (recall that in the current implementation this is the context

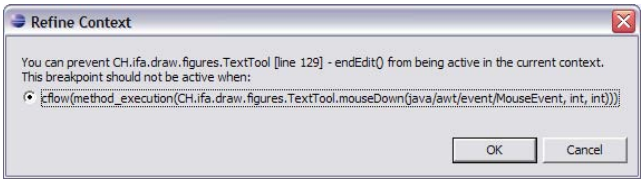

Figure 4: The "Don't Suspend if: Suggest" dialog.

in which the breakpoint is first triggered). The only menu which merits separate discussion here is the "Only Suspend if: *Suggest*" menu item which utilises slightly more complex heuristics than its counterpart in the "Don't suspend if. . . " menu.

There are three suggested pointcuts for restricting context of a breakpoint to the recorded context. The suggestions are based on two sources of information.

The first source of information is the breakpoint most recently reached prior to the recorded context. This breakpoint (which we will call b_r) is used to suggest two pointcuts. One pointcut is a copy of the control-flow condition of b_r (alternatively, if no control-flow condition is attached to b_r, then a suggested pointcut based on the most recent method on the stack when b_r was first reached is used). The other suggestion conjoins the first pointcut with one that only matches joinpoints after the line corresponding to b_r. The rationale behind these two suggestions is based on our belief that a breakpoint first reached after another breakpoint is often only of interest in the same context as the latter.

The second source of information is the call stack of the recorded context. This information is used to compose the third suggested pointcut. This pointcut matches the control-flow of the most recent method on the stack when the breakpoint was first reached. The rationale behind this suggestion is our experience that characterisation of control-flow context in terms of the call stack is most useful.

As with the "Don't Suspend if: *Suggest*" function, the current suggestion heuristics for "Only Suspend if: *Suggest*" are provisional and will likely be replaced with more sophisticated algorithms in the future.

2.4 Implementation

The Control-flow Breakpoint Debugger is implemented as

a modification to the Eclipse JDT Debugger. Additional packages and classes were added to implement both the *CBD* GUI and its supporting pointcut language, while changes to existing classes were kept to a minimum. Approximately 1400 source lines of code were added to implement the control-flow breakpoints, *BPL*, and suggestion logic. The additional dialog and context menu functionality required approximately 1600 source lines of code.

Any performance hits due to the addition of *CBD* classes are mostly associated with evaluating residual expressions when control-flow breakpoints are reached. In the current implementation, this evaluation time is—in the worst case—linearly proportional to the current call-stack depth. During our case studies, we could not observe any appreciable decrease in performance using *CBD*.

3. Evaluation and Case Studies

To evaluate the practical expressiveness of control-flow breakpoints and the adequacy of *CBD*'s graphical user interface, we performed a total of 20 case studies using the Control-flow Breakpoint Debugger, with one of the authors as the subject. Each of the case studies corresponds to a documented bug in an existing Java application.

3.1 Experimental Setup

The bugs were chosen from the bug databases of three existing Java applications—four from Battleship Wars [5], an online-gaming web application built upon the Spring Application Framework [18]; eight from version 5.3 of JHotDraw [15], a GUI-based drawing application; and eight from version 0.21.1 of ArgoUML [3], a GUI-based UML modelling application. The author participating as the subject in the case studies was familiar with the source code of Battleship Wars, but not familiar with the source code of JHotDraw or ArgoUML prior to performing the case studies. The three applications were selected based on their source code and design clarity, as well as the availability of bug databases containing suitable bugs.

Each bug was chosen by the authors of *CBD* for use in the case studies based on the clarity of the associated bug report (including preciseness of the bug description and expected correct behaviour), and the perceived ease with which the bug could be resolved—in particular, no architectural or major design changes should have been required to resolve the bug. The state of the bug (e.g. open or closed) was not taken into account during bug selection, but any solutions for resolved bugs were not viewed or included in the case studies. In addition, the bugs were not intentionally chosen to support any desired experiment results—in fact, we found it difficult to determine beforehand which bug reports would produce debugging sessions exploiting the unique features of *CBD*.

For each bug, the author participating as the subject attempted to fix the bug using Control-flow Breakpoint Debugger as the primary aid. To obtain data for later analysis, the on-screen actions of each debugging session were recorded using a screen recording utility. Additionally, on-screen notes were added in real-time by the author during the debugging session to annotate the captured video.

3.2 Results

Table 2 provides an overview of the results obtained from the 20 case studies performed. The "Bug#" column lists the bug numbers in the respective bug databases of the three applications. Under the "Dynamic Breakpoints Useful" column, a "Yes" indicates that dynamic breakpoints were used in the case study, a "Potentially" indicates that dynamic breakpoints were desired but not actually used in the case study, while a "No" indicates that no use for dynamic breakpoints could be found in the case study. Under the "CBs Adequate" column, a "Yes indicates that control-flow breakpoints were adequate for use as the desired dynamic breakpoints, a "No" indicates that some type of dynamic breakpoint other than a control-flow breakpoint was desired, while a "N/A" corresponds to a "No" in the "Dynamic Breakpoints Useful" column. Under the "CBD GUI Adequate" column, a "Yes" indicates that the programmer was able to create all desired control-flow breakpoints through the *CBD* GUI, a "No" indicates that the programmer was not able to create a desired control-flow breakpoint through the *CBD* GUI, while a "N/A" indicates that the programmer did not desire to create any dynamic breakpoints.

Our case study results support all of our 3 main claims stated in Section 1.2:

1. Dynamic breakpoints were found to be useful (in 6 cases) or potentially useful (in 3 cases) out of the 20 case studies performed. This supports our claim that situations in which the additional precision of some kind of dynamic breakpoint condition could be useful occur in practice.

2. Control-flow breakpoints were adequate in 6 out of the 9 cases in which dynamic breakpoints were found to be useful or potentially useful. This supports our claim that control-flow breakpoints are adequate in the majority of practical debugging tasks where the precision of dynamic breakpoints is desired.

3. The *CBD* GUI was exclusively used to specify the desired control-flow breakpoints in all of the 6 cases in which control-flow breakpoints were useful. This supports our claim that *CBD*'s "point-and-click" GUI supports specifying control-flow breakpoints for a majority of practical debugging scenarios.

3.2.1 Additional Observations

We now proceed to present some other interesting observations about the way *CBD* was used in our case studies.

The different types of control-flow breakpoints were used with different frequency in our case studies. In each of the 6 cases in which control-flow breakpoints were found to be useful, the following pointcut was conjoined with the breakpoint's guard condition:

```
!cflow(method_execution(...)))
```

The programmer specified and incrementally refined the context for breakpoints according to the following general pattern:

1. Create a static breakpoint at a desired location in the source code.

2. Run target program using the debugger.

3. Upon reaching the breakpoint in an undesired context, exclude the current context based on control-flow using the menu functions.

Case#	Dynamic Breakpoints Useful	CBs Adequate	*CBD* GUI Adequate	Bug#	Comments
JHotDraw					
1	No	N/A	N/A	584777	Default static breakpoints were sufficient
2	No	N/A	N/A	551103	Debugger use was limited
3	No	N/A	N/A	584776	Default static breakpoints were sufficient
4	Yes	Yes	Yes	639124	Exclusion based on current call stack was used
5	Potentially	No	N/A	639124	Desired dynamic condition based on variable values
6	Yes	Yes	Yes	639124	Exclusion based on current call stack and restriction based on recorded context were used
7	No	N/A	N/A	584772	Default static breakpoints were sufficient
8	No	N/A	N/A	514393	Debugger use was limited
Battleship Wars					
9	Yes	Yes	Yes	1	Exclusion based on current call stack was used
10	No	N/A	N/A	2	Debugger use was limited
11	No	N/A	N/A	3	Default static breakpoints were sufficient
12	No	N/A	N/A	5	Debugger use was limited
ArgoUML					
13	No	N/A	N/A	3921	Default static breakpoints were sufficient
14	Yes	Yes	Yes	3922	Exclusion based on current call stack was used
15	No	N/A	N/A	3930	Debugger use was limited
16	No	N/A	N/A	3970	Default static breakpoints were sufficient
17	Potentially	No	N/A	4097	Desired dynamic condition based on variable values
18	Yes	Yes	Yes	4145	Exclusion based on current call stack was used
19	Yes	Yes	Yes	4155	Exclusion based on current call stack was used
20	Potentially	No	N/A	4324	Desired dynamic condition based on variable values

Table 2: Overview of case study results.

4. Repeat previous step if necessary.

Refinement of breakpoint context was usually based on *exclusion* of undesired context when the undesired context was encountered, and characterisation of control-flow based on call stack methods was most often used. The "Only Suspend if: Suggest" menu function was used once in case 6.

For the 3 cases in which dynamic breakpoints were found to be potentially useful, the lack of actual use during the debugging sessions was due to the difficulty associated with manually expressing the desired dynamic conditions. In some cases, composing the required expressions would require an understanding of the source code beyond what the developer had at the time. In other cases, the required expressions would be complex to the point that it was easier to simply skip over the breakpoints when reached under undesired contexts. For example, a guard condition expressing one of the desired dynamic conditions for case 20 would be

```
ProjectManager.getManager().
                getCurrentProject().getURL() != null
  &&
ProjectManager.getManager().
                getCurrentProject().getURL().
                            getFile().length() > 0
```

Dynamic breakpoints—while frequently used—were not found to be useful in all of our case studies. There were two main reasons for this:

- Limited use of the debugger. In many cases, fixing the bug associated with the case study did not require extensive use of breakpoints (and the debugger in general). In some of the cases, code inspection was largely sufficient for discovering and fixing problematic code, while in other cases, the nature of the bug prevented effective use of the debugger—in particular, the bug associated with case 8 could not be reproduced if window focus of the program was lost during execution (the bug was a display refresh problem).
- The default static breakpoints were sufficient. In many cases, static breakpoints corresponded to the desired execution breaks for the execution paths exercised during the debugging session. Thus, no additional guard conditions were necessary for such breakpoints.

3.3 Anecdotes

To convey a sense of how *CBD* was used in some cases and failed to be useful in other cases we now present a few selected anecdotes from the case-studies. We selected the anecdotes to be a representative sampling of the different situations we encountered during the case-studies.

The motivating example presented in Section 2.1 is representative of the cases where control-flow breakpoint exclusion based on call stack methods was used in the debugging session. This was the most common scenario observed in the case studies (6 cases).

We now present two additional anecdotes from our case studies to represent the remaining interesting types of cases.

3.3.1 Anecdote of Case 6

This anecdote is derived from case 6 and demonstrates use of the "Only Suspend if: Suggest" menu function. This was the only case study in which *CBD* features other than exclusion based on call stack methods was used.

```
public class TextTool {
  public void endEdit() {
    ...
    drawing().orphan(...); // line A
    ...
  }
  public void mouseDown(MouseEvent e, int x, int y) {
    ...
    endEdit();
    ...
  }
}

public class UndoActivity {
  public void undo() {
    ...
    drawing().orphan(...);
    ...
  }
}

public class Drawing  {
  public Figure orphan(Figure f) {
    ...
    // line B
    ...
  }
}
```

Figure 5: Relevant code from case study 6.

For this anecdote, refer to the relevant JHotDraw code shown in Figure 5. We begin at the point where the programmer already used the *CBD* GUI to define a breakpoint at *line A*. This breakpoint did not have any dynamic conditions attached (i.e. it corresponded to an atomic `line_execution` pointcut).

In the debugging session, the programmer was interested in examining program state at *line B* in the same contexts in which *line A* was reached. However, *line B* is reached in both the control-flow of `TextTool.endEdit()` (corresponding to clicking away from a text box) and `UndoActivity.undo()` (corresponding to selecting the "Undo" command) while reproducing the bug. The execution within the context of `UndoActivity.undo()` was thus undesirable.

As described in Section 2, the programmer could exclude the undesirable context upon reaching it by either restricting the breakpoint based on a desirable condition from a previous execution context, or by excluding an undesirable condition from the current execution context. In this case, the programmer decided to perform the former by selecting the "Only Suspend if: *Suggest*" menu function. This caused a dialog box to be displayed with various options as shown in Figure 6. The programmer selected the first option, which resulted in the redefinition of the pointcut representing the *line B* breakpoint to:

```
line_execution(l_B)
        && after(line_execution(l_A)
        && cflow(method_execution(m)
```

where l_B and l_A are the line signatures of *line B* and *line A* respectively, and m is the method signature of `TextTool.mouseDown()`. The selected suggestion was based on the control-flow context of *line A*, as it was the most recent breakpoint reached before the breakpoint at *line B*

```
public class TextTool {
  public void mouseDown(MouseEvent e, int x, int y) {
    ...
    // line A
    ...
  }
}
```

Figure 7: Relevant code from case study 5.

(see Section 2.3 for a description of the suggestion heuristics). Future executions of *line B* in the undesirable context were then avoided for the remainder of the debugging session.

3.3.2 Anecdote of Case 5

This anecdote is derived from case 5 and is representative of cases where dynamic breakpoints were found to be potentially useful but not actually used. This anecdote conveys a sense of why the *CBD* features could not be used.

In case 5, bug reproduction required creating text boxes and clicking on the canvas multiple times in the JHotDraw GUI. Both creating text boxes and clicking on the canvas caused the execution of *line A* shown in Figure 7, but the programmer was only interested in *line A* when clicking on the canvas. Thus program suspension at *line A* in the context of creating text boxes was undesirable.

The programmer began by creating a breakpoint at *line A*. This breakpoint initially behaved as a static breakpoint and thus caused undesirable program suspension when text boxes were created. The *CBD* GUI could not be used to exclude the breakpoint from the undesirable context in this scenario for the following reasons:

- Exclusion or restriction of context based on call stack methods would not work as the call stack when creating text boxes and clicking on the canvas are identical.
- Exclusion or restriction of context based on lines of code reached before or after *line A* would not work as no other breakpoints were created.

However, program state did differ between the desirable and undesirable execution contexts. Therefore, a dynamic condition to restrict the breakpoint to the desirable context could be expressed using Java as the following:

```
(drawing().findFigureInside(x,y) instanceof TextFigure)
  || (getTypingTarget() != null)
```

This expression was derived by examining the conditional `if(...)` statements around *line A* and by examining program state in the two execution contexts. The programmer did not have the level of source code understanding required to derive this expression prior to fixing the bug, and thus was unable to attach this expression as a guard condition for the breakpoint at *line A*. Instead, he decided to simply skip over the breakpoint when reached in the undesirable context. Therefore, dynamic breakpoints were potentially useful in case 5 but were unable to be utilised during the debugging session.

The above problems encountered by the programmer in this anecdote can be generalised to the other cases in which dynamic breakpoints were potentially useful but not used. In all such cases, the *CBD* GUI could not be used to exclude

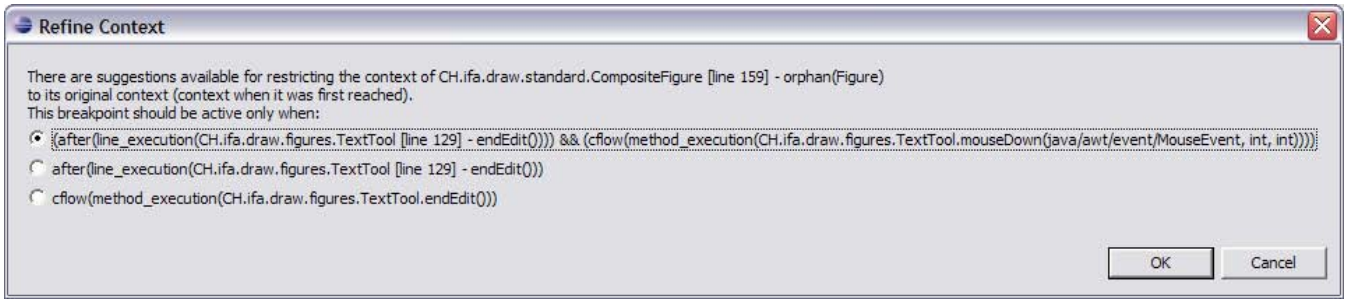

Figure 6: The "Only Suspend if: Suggest" dialog used in case 6.

the breakpoint from the undesired context for the same two reasons given in this anecdote. Also, dynamic guard conditions could be used to exclude the undesired context in all such cases, but were not used for reasons similar to those given in this anecdote.

3.4 Limitations of the Evaluation

The bugs for all 20 case studies were selected by the authors of *CBD*, and all 20 case studies were performed using one of the authors as the subject. This introduces some potential bias into our results. In particular, the experiments provide insufficient grounds to draw conclusions about the usability of *CBD* for the average developer. Nevertheless, we believe that our experiments are sufficient to support our three main claims. We explain why:

The bug selection process–while not entirely random—did not explicitly take into account the capabilities of *CBD*. As we mentioned earlier, we found that it was impossible to guess which bugs would exploit the capabilities of control-flow breakpoints or dynamic breakpoints in general. Therefore, we believe our bug-selection process is not biased with respect to selecting debugging tasks leading to situations where *CBD*'s features are more likely to be useful. Thus our results are not biased towards validating our first claim.

Our results also support the second and third claims. Note that these claims are about expressiveness as opposed to usability. In other words they are claims about whether or not it is *possible* to express the desired breakpoints using *BPL* and *CBD*'s UI respectively, not about whether the particular UI design makes this easy or intuitive for a non-author of the tool. We acknowledge that the latter will require further research.

4. Related Work

CBD has as a general goal the improvement of debugging efficiency. There are many other complementary approaches toward this goal (e.g. work on program visualisation [14]). However, most of this work is not directly related to *CBD*. In this section we discuss work directly related to Control-flow Breakpoint Debugger, and compare approaches to common problems explored by the related work.

The related work can be grouped into four general categories—dynamic breakpoint specification, trace-based aspects, aspect-oriented debugging, and automated debugging.

4.1 Dynamic Breakpoint Specification

The Control-flow Breakpoint Debugger provides a method for specifying dynamic breakpoints in program execution based on program execution flow. Work by Bruegge and Hibbard [8] in generalised path expression debuggers, as well as work by Ducassé [10] in trace queries present debugging tools which are based on pointcut languages[2] that are more expressive than ours (both languages can be used to specify breakpoints equivalent in functionality to control-flow breakpoints).

Bruegge and Hibbard extend basic path expressions, first used as a synchronisation mechanism for concurrent processes, to path rules used in a debugger. A path rule consists of an event recognition part (a generalised path expression) and an action part (a path action). The action part is taken whenever the event recognition part matches the program execution. Path rules are therefore a more powerful generalisation of traditional line breakpoints that can allow for more precise specification of both points in the program execution, and the action to be taken at the specified points.

Ducassé describes a model of program execution consisting of a sequence of events. A trace is then a particular sequence of such events, and trace queries can be written to match specific traces. These trace queries can then be used to precisely specify breakpoints in program execution.

Unlike *CBD*, these tools focus on pointcut language expressiveness and do not provide a GUI for specifying breakpoint conditions (thus requiring manual entry of expressions). In addition, the expressiveness of control-flow pointcuts was found to be largely sufficient for the particular debugging scenarios selected for our case studies, suggesting that further studies may be required to determine the practical utility of more expressive pointcuts for debugging.

4.2 Trace-Based Aspects

Our work is also related to work on expressive pointcut languages [20, 9, 6, 2]. This work investigates pointcut languages that offer increased expressiveness for dynamic pointcuts beyond AspectJ's `cflow`. Our `before` and `after` pointcuts are simple versions of these types of pointcuts.

The focus of the above mentioned work is on the application of trace-based pointcuts to general purpose aspect-oriented programming. Our work however specifically focusses on debugging. Our empirical results indicate that more experimentation may be warranted to determine the practical value of expressiveness beyond the simple `cflow` pointcut for the narrower purpose of specifying dynamic breakpoints.

[2]The languages are not referred to as pointcut languages since the terminology was not yet coined.

4.3 Aspect-Oriented Debugging

Usui and Chiba recognise that code written for debugging (such as logging code or trace messages) often involves crosscutting concerns, and that debugging code is best encapsulated in aspects [19]. Bugdel is an aspect-oriented debugging system designed to work with the AspectJ aspect-oriented language. Like Control-flow Breakpoint Debugger, it is implemented as a plugin for the Eclipse integrated development environment and provides a graphical user interface designed to avoid manual entry of pointcut expressions. Since both systems are designed for debugging, they both allow lines of code to be selected as joinpoints and both allow advice to suspend program execution.

However, Bugdel and Control-flow Breakpoint Debugger are complementary. Bugdel's main focus is in supporting the addition, modification, and removal of debugging code. As such, it gives the programmer flexibility in specifying advice bodies (e.g. the programmer can enter printing code to trace values), and even allows advice bodies to access local variables, private fields, and private methods around joinpoints. On the other hand, Bugdel is intentionally more limited in pointcut expressiveness than *CBD*, and does not support dynamic pointcuts.

4.4 Automated Debugging

Agrawal, Demillo, and Spafford [1], describe debugging as an iterative process that consists of:

1. Finding statements that had an effect on incorrect output;
2. Selecting one of the statements and setting a breakpoint at the statement;
3. Re-executing program to reconstruct program state at that point.

One of the goals of the Control-Flow Breakpoint Debugger is to make this process faster and easier by providing an easy and intuitive method for specifying a more precise point in execution to execute up to (reducing time and effort spent manually skipping over undesired breaks in execution during Step 3). There are various other approaches that also aim to streamline this debugging process. Many debuggers such as [1], [17], and [13] automate Step 1 (in some cases, they can even find statements that can potentially lead to incorrect output). Backtracking and reverse program execution are debugging techniques that supplant Step 3 (and in some cases Step 2 as well) with steps that are intended to be faster (e.g. [1], [12], [7], and [16]). All approaches mentioned above address the same problem of streamlining the iterative debugging process, but do so by focusing on different steps of the process. *CBD* uniquely focuses on streamlining Step 3 without replacing or eliminating it. We believe that our approach is easier to implement in an existing infrastructure and easier to adopt by developers, because it is a straightforward extension of the way current tools are built and how the user works with them.

5. Conclusion

In this paper, we presented Control-flow Breakpoint Debugger and made the following three claims:

1. Situations in which the additional precision of some kind of dynamic breakpoint condition could be useful occur in practice.
2. Control-flow breakpoints are adequate in the majority of these situations.
3. The purely "point-and-click" GUI of *CBD* supports specifying control-flow breakpoints for a majority of practical debugging scenarios.

We described how our approach to designing Control-flow Breakpoint Debugger involves two parts—one part consisting of *BPL*, a pointcut language for specifying control-flow breakpoints, and the other part consisting of a graphical user interface intended to allow for specification of control-flow breakpoints without manual entry of pointcut expressions.

Finally, we conducted 20 case studies using an implementation of Control-flow Breakpoint Debugger, and obtained empirical results supporting all three claims.

6. Future Work

Possible future work may involve investigation into the usability of *CBD*, including formal user studies. Such investigation would examine aspects of the *CBD* GUI not covered in this paper. Specific areas to be investigated could include ease-of-use of the user interface for developers other than the authors, techniques for conveying pointcut information through the GUI without exposing *BPL* expressions to the user, and how different GUI designs affect the usefulness of different control-flow conditions.

Future work may also involve investigation into more advanced heuristics for suggesting pointcuts to the user. A possible research direction in this area is the incorporation of additional information into the suggestion heuristics and the use of machine learning algorithms. For example, information from execution history as well as history of user-interactions with the UI could be used as input to machine learning algorithms to help suggest likely pointcut expressions.

7. References

[1] Hiralal Agrawal, Richard A. DeMillo, and Eugene H. Spafford. Debugging with dynamic slicing and backtracking. *Software - Practice and Experience*, 23(6):589–616, 1993.

[2] C. Allan, P. Augustinov, A. Christensen, L. Hendren, S. Kuzins, O. Lhotak, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding trace matching with free variables to aspectj, 2005.

[3] ArgoUML web page. http://argouml.tigris.org.

[4] AspectJ web page. http://www.aspectj.org.

[5] Battleship Wars web page. http://members.shaw.ca/util1/.

[6] Christoph Bockisch, Mira Mezini, and Klaus Ostermann. Quantifying over dynamic properties of program execution. In Robert E. Filman, Michael Haupt, and Robert Hirschfeld, editors, *Dynamic Aspects Workshop*, pages 71–75, March 2005.

[7] Bob Boothe. Efficient algorithms for bidirectional debugging. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and*

Implementation, pages 299–310, Vancouver, British Columbia, June 18–21, 2000.

[8] B. Bruegge and P. Hibbard. Generalized path expressions: A high level debugging mechanism. In M. S. Johnson, editor, *Proceedings of the ACM Sigsoft/Sigplan Software Engineering Symposium on High-Level Debugging*, pages 34–44, Pacific Grove, CA, August 1983. Association for Computing Machinery, Association for Computing Machinery.

[9] Rémi Douence, Pascal Fradet, and Mario Südholt. Trace-based aspects. In Robert E. Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Akşit, editors, *Aspect-Oriented Software Development*, pages 201–217. Addison-Wesley, Boston, 2005.

[10] Mireille Ducassé. Coca: An automated debugger for C. In *Proceedings of the 1999 International Conference on Software Engineering (ICSE'99)*, pages 504–515, New York, May 1999. Association for Computing Machinery.

[11] Eclipse web page. http://www.eclipse.org.

[12] Stuart I. Feldman and Channing B. Brown. Igor: A system for program debugging via reversible execution. In *Workshop on Parallel and Distributed Debugging*, pages 112–123, 1988.

[13] Simon Goldsmith, Robert O'Callahan, and Alex Aiken. Relational queries over program traces. *ACM SIGPLAN Notices*, 40(10):385–402, October 2005.

[14] Sadahiro Isoda, Takao Shimomura, and Yuji Ono. VIPS: A visual debugger. *IEEE Software*, 4(3):8–19, 1987.

[15] JHotDraw web page. http://www.jhotdraw.org.

[16] Bill Lewis. Debugging backwards in time. In *Proceedings of the Fifth International Workshop on Automated Debugging (AADEBUG 2003)*, October 2003.

[17] Michael Martin, Benjamin Livshits, and Monica S. Lam. Finding application errors and security flaws using pql: a program query language.

[18] Spring Application Framework web page. http://www.springframework.org.

[19] Yoshiyuki Usui and Shigeru Chiba. Bugdel: An aspect-oriented debugging system. *Proceedings of 12th Asia-Pacific Software Engineering Conference (APSEC 2005)*, 0:790–795, 2005.

[20] Robert J. Walker and Kevin Viggers. Communication history patterns: Direct implementation of protocol specifications. Technical Report 2004-736-01, University of Calgary, February 2004.