Quantifying over Dynamic Properties of Program Execution

Christoph Bockisch Mira Mezini Klaus Ostermann
Software Technology Group
Darmstadt University of Technology, Germany
{bockisch,mezini,ostermann}@informatik.tu-darmstadt.de

ABSTRACT

In a pointcut we want to fully specify the points in the execution of a program at which an advice is to be executed. The pointcut languages of current aspect-oriented programming languages only provide limited support for specifying points in the execution that do not directly map to points in the program code. As a result, an aspect programmer has to implement logic to keep track of certain runtime properties manually. This logic is detached from the advice's pointcut. In this paper, we identify two common patterns of dynamic properties on which advice rely. We present pointcut designators that allow to declaratively specify the join points based on runtime properties in a pointcut and outline a possible implementation.

1. INTRODUCTION

Pointcut-and-advice units are used in AspectJ-like languages [9] to modularize crosscutting concerns. The advice part is a piece of code and a pointcut is a special expression specifying a set of join points at which the advice must be executed. Join points are points in the execution of a program, for example reading a field or calling a method. Each join point has two parts of context. First, there is the static context, which can be retrieved by analyzing a static represenation of the program, such as the source code or bytecode. For a "field-read" join point the static context is, among others, the field's name and the class in which the field-read expression stands. Second, there is the dynamic context, which is made up of the situation at runtime when the join point is executed. Again, using "field-read" exemplarily, the dynamic context can be the value of the field, the methods on the current call stack, or even the complete history of the program execution.

The intuition behind using aspect-oriented programming (AOP) is that a concern's implementation is well localized and the knowledge of when an advice must be executed is bundled to the advice. This implies two goals we want to achieve when writing aspect-oriented code: first we want to execute code in different contexts *implicitly*, second we want to express the knowledge of these contexts *declaratively*.

The pointcut language of AspectJ has extensive support for selecting join points based on the static context. We call this pointcuts based on *static properties*. In addition, it also provides pointcut designators (PCD) for selecting join points by their dynamic context - pointcuts based on *dynamic properties*. Using the `cflow` or `cflowbelow` PCDs one can specify the methods that must be on the call stack when the join point is executed. With the PCDs `target`, `this` and `args` one can specify the dynamic type of the receiver object, active object or argument objects. However, there are more dynamic properties that can be relevant for the execution of an advice at runtime.

In this paper we give examples of aspects that show why a better support for pointcuts based on dynamic properties is needed. They show that it is not possible to specify the *precise* set of join points *declaratively* with current pointcut languages. Additionally, the relevant parts of the join points' dynamic context must be accessed programma-tically. For the given examples we have implemented aspects in Alpha [1, 11], an experimental aspect-oriented language that allows pointcuts to reason over dynamic properties of a program. It is possible in Alpha to define new pointcut designators in a declarative way. We present the resulting PCDs in a pseudo AspectJ notation, so that it is not necessary to introduce Alpha in detail in this paper.

The remainder of this paper is organized as follows. In the next section we show two examples of aspects which heavily rely on dynamic properties. We discuss how these aspects can be implemented in current aspect-oriented languages. Section 3 defines a notation for extended dynamic pointcut designators in an AspectJ-like pointcut language. A conclusion is given in section 4. Section 5 presents our ongoing work as well as some related work.

2. CURRENT STATE

In this section, we present two examples of aspects for which a pointcut must quantify over complex dynamic properties. We discuss an implementation for those examples in a conventional aspect oriented language where aspects are active globally and during the whole program execution, and in alternative languages where aspects can be deployed dynamically. This discussion shows that neither of them satisfies our goals.

The first example is a text editor application (Fig. 1). At any time, the editor has at most one document opened. Documents can be created, edited and saved. For this application, we want to write an aspect that prevents the quitting of the application and the creation of a new document when the current document is in a dirty state, i.e., there are unsaved changes.

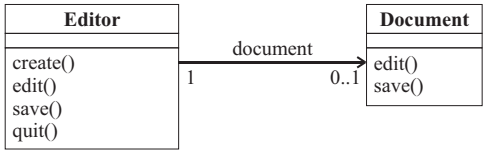

Figure 1: Base code for the editor example.

A natural description of the pointcut for this problem is: "calls to the methods `create()` or `quit()`, when there has been a previous call to the method `edit()` but no call to `save()` or `create()` since then". Fig. 2 illustrates this.

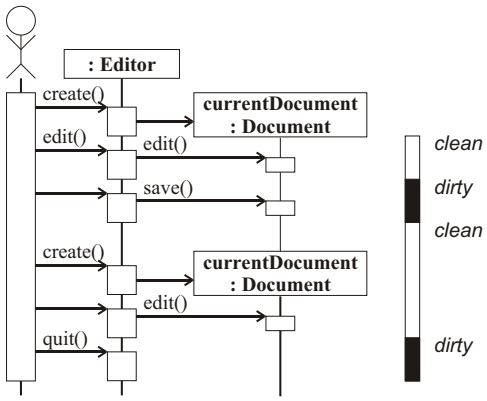

Figure 2: A possible sequence of method calls in the editor example. The last call to `quit()` should be prevented because the document is dirty at this time.

The second example is a graphical editor (Fig. 3). The program has a hierarchy of `Figure` classes that can be displayed by an instance of the class `Display`. To reflect the current state of the `Figure` objects on a `Display`, we want to define an advice calling the `Display`'s method `draw()` with the pointcut: "calls to a setter method of a `Figure` to which a `Display` points" (see Fig. 4).

2.1 AspectJ

For the discussion of a conventional aspect-oriented language we exemplarily use the AspectJ language [9]. This uses static weaving, i.e., it weaves the aspects into the program at pre-runtime. We call this "static deployment" as opposed to "dynamic deployment" discussed in the next subsection.

A possible AspectJ implementation for the editor example is given in Listing 1. The intended pointcut has a static part, namely call-instructions to the methods `quit()` or `create()`, and a dynamic part. For the dynamic part the history of the program execution must be accessed to decide

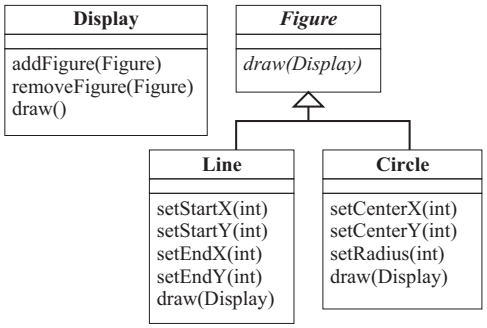

Figure 3: Base code for the display updating example.

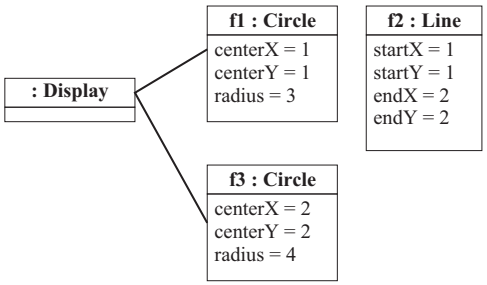

Figure 4: A possible configuration of objects in the display updating example. Changes to `f1` and `f3` should trigger a call to the `Display`'s `draw()` method.

whether there have been calls to the method `edit()` since the last call to `save()`.

AspectJ's pointcut language only allows to involve that part of the history which is still on the call stack, i.e., by the pointcut designators `cflow` and `cflowbelow`. However, in our example the relevant calls to `edit()` or `save()` are not, in general, on the stack when the method `quit()` or `create()` is called. Hence, we store this part of the history in a field of the aspect (`documentDirty`) which is maintained by two separate advice. The one advice setting the state to "dirty" after the `edit()` method has been called. And the other one setting the state to "clean" after one of the methods `save()` or `create()` have been called.

The pointcut for the advice of preventing the disposal of dirty documents is defined to match each call to `quit()` and `create()`, which is the static part. As the dynamic part an `if` pointcut designator is added. This one checks the state of the field `documentDirty` so that the pointcut only matches when the document is indeed dirty.

```
1  public aspect PreventDirtyDispose {
2    private static boolean documentDirty;
3    pointcut makeDocumentDirty():
4      call(void Editor.edit());
5    pointcut makeDocumentClean():
6      call(void Editor.save()) ||
7      call(void Editor.create());
8    after(): makeDocumentDirty() {
9      documentDirty = true;
10   }
11   after(): makeDocumentClean() {
12     documentDirty = false;
```

```
  }
  pointcut disposeDocument():
    (call(void Editor.quit()) ||
      call(void Editor.create())) &&
    if(documentDirty);
  void around(): disposeDocument() {
    // prevent quitting or creating a new document
  }
}
```

Listing 1: An AspectJ aspect that prevents the disposal of dirty documents.

By using advice to keep book of the execution history, the aspect of preventing the disposal of a dirty document is still localized. However, the knowledge of when the document is dirty is spread over several advice. As a result, it is not specified in a declarative manner but uses conditional logic.

For the second example, the intended pointcut also has a static and a dynamic part. The static part says that calls to setter methods on `Figure` objects are selected, and the dynamic part says that the `Figure` objects must be reachable from a `Display` object. The only dynamic property of the target object we can specify in AspectJ is its dynamic type. This is done by using the `target` pointcut designator (similarly there are the designators `this` refering to the active object and `args` refering to arguments of the join point). But we can not decide via the object type whether the target is reachable from a `Display` object, or not.

The display update example is basically an instance of the observer pattern [4] for which [6] presents an abstract implementation in AspectJ. Listing 2 shows a concrete adaptation that implements the presented example. The code for implementing the updating behavior is localized in the aspect.

A pointcut is specified that matches each potential join point, i.e., each call to a method whose name starts with `set` on a `Figure`. But actually the call to a setter yields a join point only if the target is an instance of `Figure` that can be reached from a `Display` object. Therefore, an `if` pointcut designator is added to specify the intended dynamic property. To make the dynamic property of "reachability" accessible to the pointcut we maintain a mapping between `Figure` and `Display` objects by separate advice.

```
public aspect DisplayUpdate {
  static private Hashtable perFigureDisplays
    = new Hashtable();
  static private Set getDisplays(Figure figure) {
    Set result = (Set) perFigureDisplays.
      get(figure);
    if(result == null) {
      result = new HashSet();
      perFigureDisplays.put(figure, result);
    }
    return result;
  }
  pointcut addFigure(Figure f, Display d) :
    call(void Display.addFigure(Figure)) &&
    args(f) && target(d);
  after(Figure f, Display d): addFigure(f, d) {
    getDisplays(f).add(d);
  }
  pointcut removeFigure(Figure f, Display d) :
    call(void Display.removeFigure(Figure)) &&
    args(f) && target(d);
  after(Figure f, Display d):
    removeFigure(f, d) {
    getDisplays(f).remove(d);
  }
  pointcut change(Figure f) :
    call(* Figure.set*(..)) &&
    target(f) && if(!getDisplays(f).isEmpty());
  after(Figure f): change(f) {
    Iterator iterator =
      getDisplays(f).iterator();
    while(iterator.hasNext()) {
      ((Display) iterator.next()).draw();
    }
  }
}
```

Listing 2: An AspectJ aspect that implements the display updating concern.

As in the editor example, the aspect of display updating is localized, but it is not possible to specify the intended pointcut declaratively. The pointcut is even harder to understand because the intended dynamic property is only accessed indirectly by the maintenance logic. The question "is an object `a` reachable from another object `b`?" is not answered by investigating the object heap. It is rather approximated by investigating method calls, namely calls to `addFigure()` and `removeFigure()`.

2.2 Other Approaches

In AspectJ the pointcuts are evaluated at weave-time. For pointcuts relying on static properties the evaluation results in points in the program code that directly correspond to the specified points in the execution. As a result the advice can be woven there. For pointcuts that specify dynamic properties, the evaluation only results in potential join points. AspectJ generates code that checks if the dynamic properties are satisfied there. The advice is only executed if the check succeeds [8]. This is also the case for the `if` pointcut designator used in the last subsection.

There are other approaches of aspect-oriented languages that allow aspects to be deployed at runtime [10, 2] or even to deploy aspects for single objects [3, 12] (we will refer to this as instance-local deployment). In such languages, it is possible to define pointcuts that only depend on static properties and deploy/undeploy an appropriate advice when the dynamic property becomes satisfied respectively unsatisfied. In the editor example, we would write the pointcut `call(* Editor.create()) || call(void Editor.quit())`. The advice would be deployed after the method `edit()` has been called and undeployed after a call to `save()` or `create()`. A more general description of this approach is given in Morphing Aspects [5].

With instance-local deployment, we can write the pointcut for the display update example by only specifying the static part of it, such as `call(void Figure.set*(..))`. An advice with this pointcut is deployed for `Figure` objects that are added to a `Display` and undpeloyed for each `Figure` that is removed from a `Display`.

Similar to the presented examples, the deployment can also be realized as an advice. However, when using dynamic de-

ployment, the knowledge of when the advice really must be executed is also not expressed declaratively as it was postulated as a goal of AOP ealier in this paper. The lack of runtime checks before executing the advice leads to improved performance [7], but does not help writing aspect-oriented code more clearly, at least for the two presented problems.

3. EXTENDED POINTCUT DESIGNATORS FOR DYNAMIC PROPERTIES

In this section, we show how to implement the examples from section 2 in Alpha [1, 11], a toy aspect-oriented language, where pointcuts can be specified based on a rich set of dynamic properties. To accomplish this, facts are generated in a Prolog [13] database at relevant join points[1]. This database contains facts about (1) the abstract syntax tree of the program, (2) the type hierarchy, (3) a complete history of the execution trace up to the current point in execution, and (4) the current content of the heap. Pointcuts are written as Prolog queries which provides us with functional abstraction for pointcut designators. Everytime a fact is added to the database, all pointcuts are evaluated. When the join point matches a pointcut, the associated advice is executed. All this combined allows us to define high-level pointcut designators that rely on a rich set of static and dynamic properties of join points.

AspectJ is widely known and thus we present the resulting high-level pointcut designators as suggestions for extensions to AspectJ's pointcut language. By doing so we can omit an introduction to Alpha which is not necessary for argumenting the need of better pointcut designators. These extensions are tailor-made for the presented examples and are not useful in general. The general pointcut designators implemented as extensions to Alpha can be downloaded from [1].

To specify the control flow in which join points can occur it must be possible to specify when the control flow is entered and when it is exited. To stay conform with AspectJ, one pointcut is specified that describes the join points at which the control flow is entered and one pointcut can be specified at which the control flow is left. As opposed to the PCDs `cflow` and `cflowbelow` both pointcuts generally are distinct. Further, the points in the execution to be specified can be just before a join point or just after a join point (similar to the difference of a `before` and an `after` advice), see Fig. 5. Thus, we suggest to add the following pointcut designators to AspectJ.

```
afterstart(<pointcut>)
afterend(<pointcut>)
beforestart(<pointcut>)
beforeend(<pointcut>)
```

With these extensions, the pointcut for the editor example can be specified as in Listing 3. The first part of the pointcut specifies method calls that lead to the disposal of the current document, as in Listing 1. The `afterend` and `beforeend` PCDs declare that join points can only match

[1]Alpha uses static optimization techniques to find the smallest possible set of join point shadows. This process can be compared to the evaluation of pointcuts by the AspectJ weaver [8].

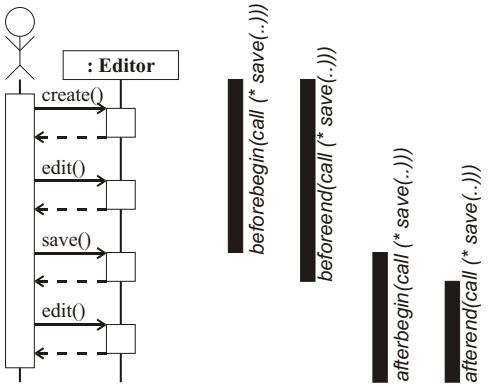

Figure 5: Join points specified by the new control flow based pointcut designators.

after the current document has been put into a dirty state (i.e., the method `edit()` has been called), and before the document has been made clean. Of course, we can not look into the future to see whether the `beforeend` pointcut will eventually match. Thus the latter PCD is implemented in a way that the complete pointcut only matches when the `afterend` pointcut has already been matched during the execution, but the `beforeend` pointcut has not matched since then or never at all. If any of these pointcuts has been matched several times in the execution, only the last time is taken into account. The right ruler in Fig. 2 is black for the time spans at which both conditions are satisfied. Join points can only occur during those time spans.

```
pointcut disposeDirtyDocument():
  (call(void Editor.quit()) ||
    call(void Editor.create())) &&
  afterend(call(void Editor.edit()) &&
  beforeend(call(void Editor.create()) ||
            call(void Editor.save()));
```

Listing 3: A pointcut for the editor example using the extended pointcut language.

For object-graph based pointcuts we have identified the need for a pointcut designator based on the "reachability" property for the target in the context of the join point. We suggest the pointcut designator `targetreachable(<type>)` to select join points where the target is reachable from an object of type `<type>`. With this designator we can define the pointcut for the display update example declaratively. As is possible with the `target` PCD in AspectJ, values from the join point's context can be bound with the `targetreachable` pointcut designator. Unlike in AspectJ it is, however, possible that several `Display` object have the property that the `Figure` is reachable from them. In this case the associated advice is called once for each value in the context.

```
pointcut changeDisplayedFigure():
  call(void Figure.set*(..)) &&
  targetreachable(Display d);
```

A similar pointcut designator can be added to specify the reachability property for the active object, i.e., `thisreachable(<type>)`. In our prototype the reachability property can also be specified for argument objects. But we will not

present this here because this PCD requires a different syntax than `targetreachable` and `thisreachable`.

4. CONCLUSION

With the possibility of defining *precise* pointcuts in a *declarative* way as presented in the last section, the meaning of an advice becomes more clear. In the presented AspectJ implementation of the aspects in section 2 we used the `if` PCD to refer to dynamic properties in the pointcuts. Thus, you have to understand the helper-advice for maintaining the state accessed in the `if` PCD before you can understand which join points are selected.

What's more the presented examples each represent a pattern that occurs repeatedly. For example the specification of a control flow as in the editor example is also used by other concerns. Let's assume we want to add an aspect for recording macros to the editor. The user can start and stop the recording by pressing a button which results in a call to the method `startMacro()` and `stopMacro()` accordingly. Between two such method calls, each call to `edit()` must be recorded. The problem is very similar to the prevention of disposing a dirty document, but we have to implement the complete logic for keeping track of the execution history, over again.

We used AspectJ's pointcut language to describe our extensions to the pointcut language because AspectJ is widely known. However, we suggest to exetend the pointcut languages not only of conventional AO lanuages with globally deployed aspects. The suggested extensions are complementary to the deployment mechanism. Steamloom [3], for example, uses an AspectJ-like pointcut language, too, including the PCDs based on dynamic properties that already exist in AspectJ, such as `cflow`.

5. ONGOING AND RELATED WORK

We are currently classifying properties on which pointcuts can be founded. The properties are rated based on the complexity needed for an efficient implementation. We are also exploring the possibility to implement pointcuts based on dynamic properties in an execution environment with dynamic and instance-local deployment that is being developed at our group (Steamloom [3]).

[14] presents an extension to AspectJ where temporal relations between events can be specified in a pointcut that must be true to select a join point at runtime. The relations are defined using context free grammars. The authors of [14] do not, however, analyze the need for expressing dynamic properties in pointcuts, in general. We see this extension as a possible notation for pointcut designators based on the execution history as well as a possible implementation of such PCDs.

6. REFERENCES

[1] Alpha project. http://www.st.informatik.tu-darmstadt.de/pages/projects/alpha/.

[2] AspectS homepage, 2003. http://www.prakinf.tuilmenau.de/ hirsch/Projects/Squeak/AspectS/.

[3] C. Bockisch, M. Haupt, M. Mezini, and K. Ostermann. Virtual Machine Support for Dynamic Join Points. In *AOSD 2004 Proceedings*. ACM Press, 2004.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison Wesley, 1995.

[5] S. Hanenberg, R. Hirschfeld, and R. Unland. Morphing aspects: incompletely woven aspects and continuous weaving. In *AOSD*, pages 46–55, 2004.

[6] J. Hannemann and G. Kiczales. Design pattern implementation in Java and AspectJ. In *Proceedings OOPSLA '02. ACM SIGPLAN Notices 37(11)*, pages 161–173. ACM, 2002.

[7] M. Haupt and M. Mezini. Micro-Measurements for Dynamic Aspect-Oriented Systems. In M. Weske and P. Liggesmeyer, editors, *Proc. Net.ObjectDays 2004*, volume 3263 of *LNCS*. Springer, 2004.

[8] E. Hilsdale and J. Hugunin. Advice Weaving in AspectJ. In *Proc. of AOSD'04*. ACM Press, 2004.

[9] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proceedings of ECOOP '01*, 2001.

[10] M. Mezini and K. Ostermann. Conquering aspects with Caesar. In *Proceedings Conference on Aspect-Oriented Software Development (AOSD) '03*, pages 90–99. ACM, 2003.

[11] K. Ostermann, M. Mezini, and C. Bockisch. Expressive pointcuts for increased modularity. In *Proceedings ECOOP '05. to appear*. Springer.

[12] K. Sakurai, H. Masuhara, N. Ubayashi, S. Matsuura, and S. Komiya. Association aspects. In *Proc. of AOSD'04*. ACM Press, 2004.

[13] L. Sterling and E. Shapiro. *The Art of Prolog*. MIT Press, 1994.

[14] R. J. Walker and K. Viggers. Implementing protocols via declarative event patterns. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE-12)*, 2004.