

# Improving Dynamic Data Analysis with Aspect-Oriented Programming

Thomas Gschwind      Johann Oberleitner  
Abteilung für Verteilte Systeme  
Institut für Informationssysteme  
Technische Universität Wien  
Argentinerstraße 8/E1841  
A-1040 Wien, Austria  
{tom,joe}@infosys.tuwien.ac.at

## Abstract

*In this paper we present a new instrumentation approach to reverse engineer a given software application. Our approach is based on aspect-oriented programming and provides support for dynamic feature analysis. The advantage of our approach compared to other existing approaches is that we allow the engineer to obtain deeper insights into the program executions and to combine these insights with existing analysis techniques. As we show in this paper, this significantly reduces the time necessary to obtain viable traces of a program's execution.*

## 1. Introduction

Before an existing software application can be extended, it is necessary to have at least a partial understanding of the implementation of the application. Although approaches to completely reverse engineer an application exist [10], they pose too much overhead in many cases. Applications are typically structured into multiple components, each consisting of hundreds of lines of code. This makes it hard to identify the modules that need to be understood just by looking at the program's source code.

It is already well understood that dynamic program analysis [4, 13] simplifies reverse engineering by providing good starting points for static program analysis, especially if the program's documentation is vague or does not exist at all. Using dynamic program analysis requires the instrumentation of the original software application or its underlying runtime system to generate traces of real program executions. By studying these traces, it is typically possible to identify those parts of the program that implement the functionality of interest and hence, need to be understood.

The dynamic analysis approaches available today, however, have one or more of the following shortcomings.

- Program traces cannot be analyzed online and thus several program executions are necessary. Sometimes this requires the recompilation of the original program for each additional trace to be generated.
- Traces are not available on a per object basis and cannot be tuned to method invocations of specific object instances.
- Argument values passed to objects are not available, thus limiting the detail of analysis.

To overcome the above limitations, we have implemented ARE, A Reverse Engineering tool that runs as part of the software application to be understood. This allows engineers to introspect objects and subsequently to refine their data collection while the program is executing.

For the integration of ARE into the software application to be understood, we are using AspectJ [7] in combination with an aspect definition that is responsible for the data collection and the invocation of the ARE's analysis module. The benefit of using AspectJ is that it provides detailed access to the method calls of the application of interest including the individual arguments passed to a method. On the basis of this data, it is not only possible to generate program traces but to focus on the use of individual object instances.

To demonstrate the advantages of having such detailed information at hand, we have reverse engineered Sun Microsystems's BeanBuilder application up to the point that allowed us to add support for recording and to playing back macros, an extension that affects many of the modules provided by the BeanBuilder application.

This paper is structured as follows. Section 2 gives a brief overview of aspect-oriented programming with focus onto AspectJ. In Section 3 we explain how we use this tool

```

aspect MacroLogic {
    pointcut instrumentedListenerMethod(): within(EventListener+) && execution(public *(..));

    before(): instrumentedListenerMethod () {
        // ... record an event ...
    }

    declare parents: EventListener+ implements IMacroEventTarget;

    public void EventListener+.simulateMacroEvent(Method m, Object[] parameters) {
        // ... simulate an event ...
    }
}

```

**Figure 1. A macro-recording aspect**

to instrument the software application to be reverse engineered and how the trace data is collected. How this data can be analyzed is presented in Section 4. Section 5 shows how we have reverse-engineered and how we have extended the BeanBuilder application. Future work is presented in Section 6 and related work in Section 7. Finally, we draw our conclusions in Section 8.

## 2. AspectJ

The goal of aspect-oriented programming is to extract functionality that is scattered throughout the whole application such as logging, transaction management, or security management into a separate module [9].

Typically, such kind of functionality is referred to as an aspect of the underlying software application. Frequently cited examples for such aspects are logging, transactions or security management. Depending on the application domain, other aspects can be identified as well.

AspectJ [7] is a compiler that supports aspect-oriented programming. AspectJ is an extension of the Java programming language that is fully backward compatible to the Java language specification [6]. In addition to the functionality provided by Java, AspectJ supports the definition of aspects. An example for an aspect is given in Figure 1. This aspect implements a macro recorder for an application that we have implemented previously. This aspect shows also one of the limitations of aspect-oriented programming. An aspect that has been implemented for one application frequently cannot be reused for another application. This got apparent when we tried to use the very same aspect for the BeanBuilder application. As soon as we started recording a macro, the BeanBuilder application became useless.

The aspect consists of four definitions. The first one defining a pointcut. Pointcuts refer to specific actions that occur during the execution of a program such as the execution of a given method, the execution of an exception

handler, or the access of an object's field. A pointcut consists of a name and a set of pointcut designators that build a boolean expression to identify the join points within an execution. Pointcut designators for method execution and field access take signatures as arguments that denote the methods or the fields to be matched by the pointcut. Other designators restrict the set of the classes that the pointcut applies to. These designators take type patterns that describe these classes. Wildcards can be used within signatures to extend the definition of a pointcut to multiple method or field signatures. Type patterns also allow the use of wildcards to specify classes that have a particular naming scheme or specify classes that have a certain supertype or implement a certain interface. The pointcut of our macro aspect for instance applies to all classes that implement the `EventListener` interface. This pointcut designator is combined conjunctively with another designator that denotes the execution of a public method that has an arbitrary name and an unspecified number of arguments.

The second definition specifies advice code that has to be executed whenever the execution of the program hits a pointcut. This code may be executed either before or after the execution of the corresponding pointcut as shown in our example. In our example it is executed before the pointcut (a method call, for instance). Additionally, the advice code may be executed around the pointcut. If the advice code is put around the pointcut, it is even possible to skip the execution of the pointcut. The before advice provided in the macro contains the logic to record each method call. To provide access to arguments of a method that is cross-cut, AspectJ provides a small runtime library. Hence, it is possible to store all argument values of the method call.

AspectJ allows the extension of types by changing its list of implemented interfaces or by changing its superclass. As shown in the third definition the `declare parents` construct states that all classes that implement the `EventListener` interface should also implement

```

aspect Tracing {
    pointcut classes(): !within(at.ac.tuwien.infosys.are.*);
    pointcut allCalls(): classes() && (call(new(..)) || call(* *(..)));

    before(): allCalls() {
        if(StaticData.recorder!=null) StaticData.recorder.beforeCall(thisJoinPoint);
    }
    after(): allCalls() {
        if(StaticData.recorder!=null) StaticData.recorder.afterCall(thisJoinPoint);
    }
}

```

**Figure 2. The tracing aspect**

the `IMacroEventTarget` interface. Since this interface contains the method `simulateMacroEvent` this method has to be included in all classes that realize an advise. This can be done by putting this method in the aspect as shown by the last definition in Figure 1. AspectJ includes all regular Java fields or methods that are defined within an aspect in the classes denoted by pointcuts.

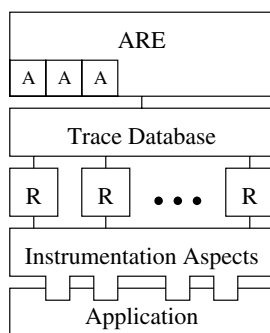
### 3. Tracing

We have used AspectJ to define a set of instrumentation aspects that add code to a given Java program to gather enough tracing information such that the program can be reverse engineered. Unlike the macro recording aspect shown in Section 2, tracing is an aspect that is well understood and can be generalized to be used in combination with any software application. A simplified version of such an aspect is shown in Figure 2. The `thisJoinPoint` variable provided by AspectJ contains meta-information about the joinpoint. In our instrumentation, we use this variable to obtain the method signatures and arguments. This allows engineers to track the function calls and the objects that are passed around by a software application (Figure 2).

As depicted in Figure 3, the aspect simply forwards each method invocation to one or more recorder modules (R). If necessary, the instrumentation aspect may be changed for instance to instrument only a part of the original application. This might interesting, for instance, if performance is of concern. Otherwise, the standard instrumentation-aspect is used. The recorder modules are responsible for filtering and recording the calls of interest and putting their traces into a trace database. The top layer consists of the ARE tool itself which is responsible for the data-analysis and which can be extended with custom analysis tools (A).

Using AspectJ to obtain the trace-data from the application to be understood has the following advantages.

- The aspect is woven into the program autonomously.
- AspectJ does not turn off Java’s just-in-time compiler as has to be done for instrumentation approaches using Java’s Virtual Machine Debug Interface (JVMDI) [15].
- It provides a mapping from the joinpoints specified to the application’s source files and line numbers.
- AspectJ allows to access the parameters that are passed as part of a function call and thus allows the traces to be tuned to specific object instances.



**Figure 3. Architecture of ARE**

A minor drawback of AspectJ, however, is that the aspect woven into the application remains unchanged for the whole program execution. This is, why the aspect generates data for each method invocation, and subsequently is processed by a recorder module that is responsible for filtering the method invocations the engineer is interested in. The recorder, however, can be customized during run-time allowing engineers to enable tracing only during the execution of those functions, objects, or object instances, he is interested in.

Since the current version of AspectJ uses the source code of an application to weave in the aspects, our approach requires the availability of the applications source code. Future versions of AspectJ, however, will be able to operate

directly on the application's byte code and will not require the availability of the application's source code [8].

## 4. Analysis

While the function calls can be obtained from a tool that generates an application's call graph as well, our tool has the advantage that it allows to track the objects that are being passed around between the different modules of a software application.

In addition to the standard analysis of the program execution such as sequence diagrams and mapping method calls to source files, ARE supports the inspection and manipulation of objects during run-time, gives access to parameters of the individual function calls and hence is able to map reflective (dynamic) method calls into a human readable form.

### 4.1. Execution Traces

With the tracing approach we have presented every method invocation is intercepted and may be recorded. Since recording each method invocation would yield too much data to be analyzed, the recorder may be customized using a filter expression, such as the expression shown in Figure 4. This filter expression instructs the recorder to record only those function calls that are directed to an object of the type `javax.swing.JMenuBar`.

The trace file obtained from the Bean Builder application using this filter expression is shown in Figure 5. Each line in this trace represents a method call, indicated with a “{“ after the method's signature, or the return from a method call, indicated with a “}” before the method's signature. For brevity reasons, some method calls have been omitted from the figure; these are marked with “. . .”. As can be easily identified from the trace, the bean builder uses a separate function for the construction of each menu. As explained in Section 5, using this trace, we were able to add the menu for the macro recorder to the BeanBuilder application.

### 4.2. Object Inspection and Manipulation

In the previous subsection, we have already presented standard analysis techniques that are present in almost all of today's reverse engineering tools. Since our analysis component is executed as part of the original application, we can inspect and manipulate objects of the application in a manner similar as provided by many debugging environments.

Whenever the engineer is interested in one of the method invocations presented in the tracing output, he may click onto that method to obtain additional information about its source and target objects as well as the parameters. ARE also displays the source file and the line number where the

method in question has been declared. This allows engineers to obtain deeper insights and to identify how an object is being used. Additionally, ARE allows engineers to assign names to these objects and parameters which will be shown in UML Sequence Diagrams [5] and other inspection windows. These names may be used in subsequently defined filter expressions facilitating further refinement.

### 4.3. Object Monitoring and Tracking

Since our analysis tool is running as part of the original application and the engineer can get hold of various object instances, we also provide a means to monitor and track the use of a given object instance. Objects can be monitored from different perspectives:

1. The use of this object from other objects can be monitored which is useful if the interface of the object needs to be understood.
2. The calls from the object to other objects can be recorded which is necessary if the object's implementation needs to be understood.
3. How and when this object is being passed from one object to another object. This is useful to track how an instance is passed around during the execution of the program.

It is also possible to use a combination of the above choices. An example for monitoring how an object is being used will be shown in Section 5.

### 4.4. Accessing a Method's Parameters

Although, we initially thought that accessing the parameters that are being passed to a method call is just a nice little gimmick, we were surprised that this functionality was of crucial importance for the reverse-engineering of the BeanBuilder application. The BeanBuilder application uses Java's reflection API [16] (Java's Dynamic Method Invocation Interface).

Java's Reflection API allows a method call to be constructed at runtime, thus enabling to invoke a method of an object whose interface was unknown at implementation time. Such a method call is constructed by obtaining a reference to the method to be invoked. Finally, the method is invoked by calling the method reference's `invoke` method. The arguments of this method are the object to be passed as this parameter and an array of objects to be passed as arguments of the method to be invoked.

Figure 6 shows a standard trace of a reflective method invocation as well as the same trace enriched with the additional data gained from our instrumentation approach. In

```

public class EventRecorder extends AbstractRecorder {
    public EventRecorder(Properties props) {}

    protected boolean filter(JoinPoint joinPoint) {
        return joinPoint.getTarget() instanceof javax.swing.JMenuBar;
    }
}

```

**Figure 4. MenuBar filter expression**

```

void com.sun.java.swing.ui.CommonMenuBar.configureMenu() {
    ...
    JMenuItem beantest.BeanMainMenu.createIconsMenu(String, char) {
        ...
    } JMenuItem beantest.BeanMainMenu.createIconsMenu(String, char)
    JMenuItem javax.swing.JMenuBar.add(JMenuItem) {
    } JMenuItem javax.swing.JMenuBar.add(JMenuItem)
    JMenuItem beantest.BeanMainMenu.createHelpMenu(String, char) {
        ...
    } JMenuItem beantest.BeanMainMenu.createHelpMenu(String, char)
    JMenuItem javax.swing.JMenuBar.add(JMenuItem) {
    } JMenuItem javax.swing.JMenuBar.add(JMenuItem)
    JMenuItem javax.swing.JMenuBar.add(JMenuItem) {
    } JMenuItem javax.swing.JMenuBar.add(JMenuItem)
} void com.sun.java.swing.ui.CommonMenuBar.configureMenu()

```

**Figure 5. Trace obtained with MenuBar filter expression**

#### Simple Reflective Method Call Trace

```
Object java.lang.reflect.Method.invoke(Object, Object[]) {
```

#### Reflective Method Call Trace Augmented With Parameter Information

```
Object java.lang.reflect.Method.invoke(Object, Object[]) {
- source=beantest.property.PropertyTableModel@4bb369
- target=public java.awt.Color java.awt.Component.getBackground()
- arg[0]=javax.swing.JButton[,17,25,104x67,layout=...]
- arg[1]=[Ljava.lang.Object;@95f290
```

**Figure 6. Trace of a reflective method invocation**

the original trace, we can only identify that a method has been invoked using reflection. Only using runtime data in the form of the arguments passed to the `invoke` method, we are able to identify the real target object (a `JButton` object) and the method to be called (`getBackground`).

## 5. Evaluation

To evaluate our approach, we have taken Sun Microsystems's `BeanBuilder` [17] application and tried to extend it solely on the information gathered using ARE, our reverse-engineering tool. The `Bean Development Kit` is a bean composition environment provided that can be used as a testbed for newly implemented `JavaBeans` components. It allows developers to instantiate such components, to change various properties such as the component's color or font, and to connect them to other components. Additionally, this toolkit is popular among researchers to test extensions of the `JavaBeans` component model.

Before we could start reverse-engineering the `BeanBuilder` application, we had to instrument it with our tracing and analysis components. We only had to replace the Java compiler used by `BeanBuilder`'s build system with the `AspectJ` compiler and had to add our instrumentation aspect to the list of source files (a change of 3 lines in total). In future versions, this can be even more simplified by providing an `AspectJ` wrapper that behaves like the Java compiler itself and transparently adds our instrumentation aspect to the list of source files. After recompilation the application was instrumented and we were able to use ARE's analysis component running as part of the original application.

The first extension adds support for a user-customizable menu where users may add their own tools that they frequently use. This extension was mainly used for us to get familiar with the use of our own tool and with the use of `AspectJ`. Since we did not want to change the source-code of the application itself, we have again used `AspectJ` to weave these extensions into the `BeanBuilder` application.

Using `AspectJ` for the instrumentation and for the extension of the `BeanBuilder` application does not pose a problem since the two aspects do not negatively influence each other. Generally speaking, using our approach in combination with other applications making use of `AspectJ` should only pose minor problems since the tracing aspect does not change the flow of execution of the original application. Depending on the order the aspects are applied our approach might not be able to identify some of the control flow changes imposed by the aspects employed by the original application.

To find out how the menu has to be woven into the application, we used a `MenuBarRecorder` that records all invocations of a `JMenuBar` object. Using this recorder, we have derived the execution trace shown in Figure 5. On

the basis of this trace, we were able to formulate the aspect shown in Figure 7.

The other extension adds support for recording macros and playing back user-defined macros, functionality which is frequently desired but missing not only in the `BeanBuilder` application. This extension also uses the previous extension to display menu items to record and to play back macros that have been recorded and again has been implemented as an aspect. The initial aspect definition that we have already used in a different application is shown in Figure 1.

Our original macro recorder is based on the fact that Java event handlers usually implement the interface `java.util.EventListener`. The macro recorder uses an aspect for the instrumentation of Java event handlers. Hence, we can apply this aspect to all descendants of the `EventListener` interface. Statements for recording the method name and the arguments of an event handler are woven into each public event handler method. In addition each event handler class gets changed and implements an interface that provides a method for the invocation of a previously recorded event handler call with the corresponding parameters.

Recording every call of an event handler for recording the macro, however, produces too much data and also records internal events that must not be played back. To get hold of the events to be recorded by our macro recorder, we have used ARE to obtain a trace of all the events that occur when a new button `JavaBean` is created with the `BeanBuilder`. One such event recorded is shown in Figure 8 as UML Sequence Diagram. The event starts with the invocation of `doAddBean` within the `InstantiationHandler`. Subsequently, a new component is instantiated (`getNewPaletteBean`) and added to the `ObjectHolder` and the `TreePanel` displaying the instantiated components.

The `InstantiationHandler` is not shown as the first object, since it was already used by other events that are displayed in the UML Sequence Diagram Window. Calls that have not been recorded by ARE are indicated with the label (`missing calls`) in the diagram. A bug, however, shown in the Diagram is that we do not yet draw the arrows indicating self-calls correctly. This will be fixed in another version of ARE.

To identify how properties of a `JavaBean` are being changed with the `BeanBuilder`, we have used the same approach as for the creation of objects. We simply recorded the events generated for changing the button's background color. After looking at the trace, it was obvious that none of the events recorded was responsible for changing the `JavaBean`'s property. As our next approach, we instructed ARE to record all method invocations of the button `JavaBean` we have instantiated previously. The `JavaBean` instance to be

```

aspect MacroInMain {
  before(JMenu menu) : args(menu) && call(JMenuBar.add(JMenu)) {
    if("Help".equals(menu.getText())) {
      JMenu macroMenu=new JMenu("Macros");
      macroMenu.add(new StartRecording());
      macroMenu.add(new StopRecording());
      ((JMenuBar) thisJoinPoint.getTarget()).add(macroMenu);
    }
  }
  ...
}

```

Figure 7. The MenuBar aspect

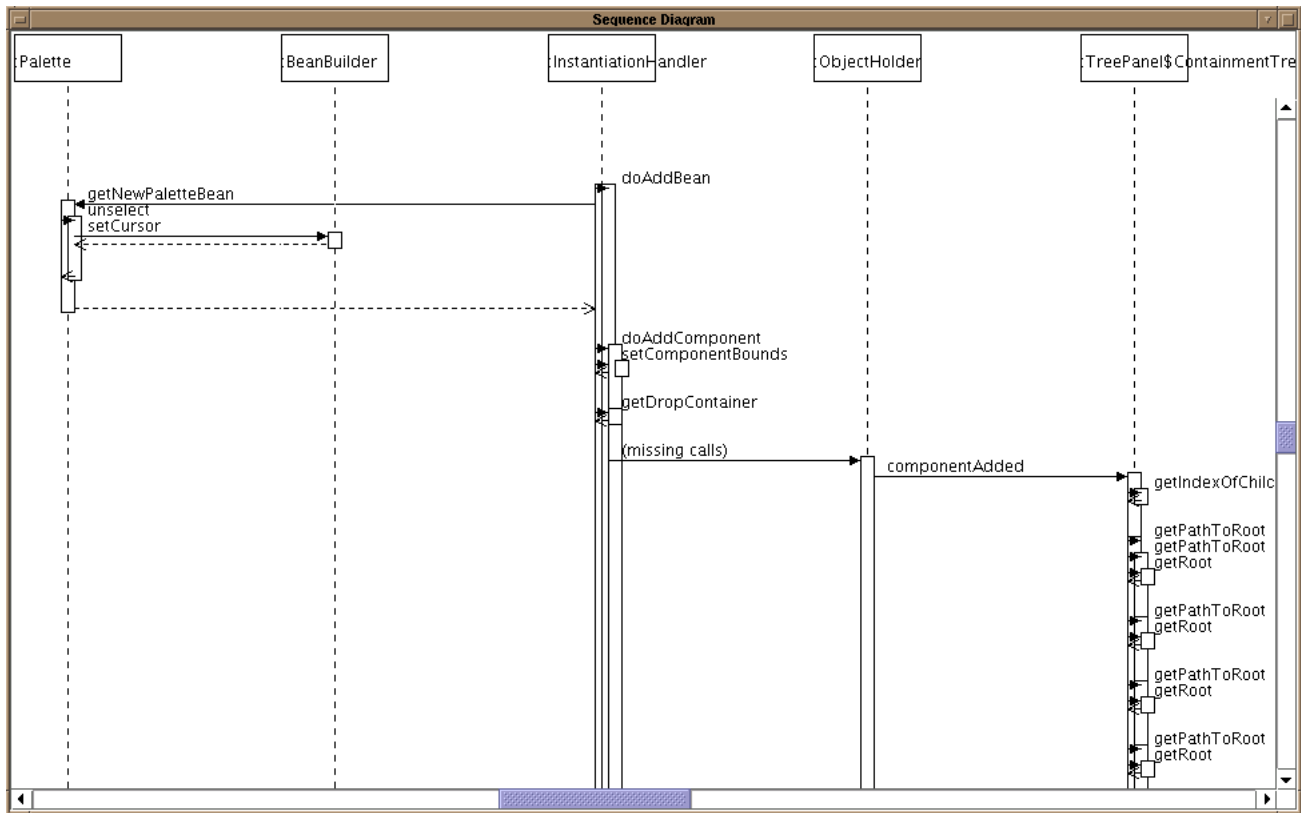


Figure 8. Creation of a new JavaBeans component

```

public class RefinedReflectionRecorder extends AbstractRecorder {
  public ReflectionRecorder(Properties props) {}

  protected boolean filter(JoinPoint joinPoint) {
    return joinPoint.getTarget() instanceof java.lang.reflect.Method &&
           joinPoint.getArgs()[1] == StaticData.variables.get("button");
  }
}

```

Figure 9. A refined filter to trace reflective method calls

monitored was identified quickly and on the basis of the `ObjectHolder` shown in the UML Sequence Diagram. Additionally, we have assigned the name “button” to the `JavaBean` instance to allow us to reference this instance in our filter expression. To our surprise, the method to change the button’s background has not been called at all.

After reconsidering the problem, we came to the conclusion that the `BeanBuilder` application must use reflection to change the `Button`’s background. The `Button` is a component that is not an integral part of the `BeanBuilder` and may have been added to the list of available components afterwards, hence the `BeanBuilder` does not know the interfaces provided and has to use the reflection API. We verified this assumption by tracing all reflective method calls where the button object was passed as the target object. This filter expression is shown in Figure 9. It also shows how the “button” `JavaBean` can be referenced by a filter expression.

On the basis of this data, we were able to adapt our macro recording aspect to record `doAddBean` methods and reflective method invocations of objects placed into the `ObjectHolder`.

## 6. Future Work

Currently, ARE does not provide any kind of static program analysis. Hence, in some cases, the engineer will have to revert to using the source code to gain a deeper understanding of the software application at hand. By adding static program analysis, we could automatically generate sequence diagrams for specific methods not yet executed. Additionally, these diagrams could be augmented with information gained from the dynamic program analysis such as argument values.

So far, ARE does not support any kind of concept analysis. With this kind of support, ARE could help the engineer to generate a matrix specifying which methods and classes are used for the implementation of which feature. Using this matrix, an engineer can identify the interface of a certain feature and whether it is encapsulated within its own module.

`AspectJ` is the most popular aspect-oriented tool available today. However, for the Java programming language other aspect-oriented tools such as `Hyper/J` [12] have been developed that are different from `AspectJ`. We have to investigate if the tools provide a higher number of potential instrumentation points to allow for a more fine-grained analysis. `Compost` [2] is a composition system that works directly on the abstract syntax tree of Java source code files allowing for a finer grained instrumentation. `Compost`, however, is more difficult to use than `AspectJ` and we do not know if the instrumentation with `Compost` can be accomplished as easily as with `AspectJ`.

Aspect-oriented programming is not restricted to the Java programming language. According to [1] a variety of projects is concerned with the implementation of aspect-oriented tools for programming languages such as C, C++, C#, Ruby, Smalltalk, and even Perl. `AspectC++` [11] has already been used for program instrumentation in the area of debugging and monitoring. Hence, it should be possible to use some of these compilers to support the corresponding languages in future versions of ARE.

## 7. Related Work

Dynamic analysis of applications is used for program comprehension, dynamic optimization and application monitoring. Several tools and technologies have been developed that are related to our work. These tools, however, differ in the area of analysis and the type of instrumentation that they use.

`Software Reconnaissance` [18] uses test cases as probes to locate code for a particular product feature. The program is instrumented similar to instrumentation for test coverage. Afterwards, two different execution runs are started. In the first run a few test cases are applied that exhibit the desired feature. In the second run other test cases are used that do not use this feature. The difference of the trace files generated by the instrumented code shows which code has been executed by a test for a particular feature. Hence, it is possible to build a mapping between features and code. An extension of `Reconnaissance` that supports concept analysis was presented in [4]. Concept analysis is used to identify the most feature-specific subprograms among all executed subprograms. A static analysis uses this subprogram to identify additional feature-specific subprograms along a dependency graph. While `Reconnaissance` provides support for well established analysis techniques, it does not allow engineers to gain as much information as provided by ARE.

`BEE++` [3] is a C++ based object-oriented framework for the dynamic analysis of distributed systems. `BEE++` considers the execution of a distributed system as a stream of events. Hence, `BEE++` allows the customization of events and event views. Event and event view customization rely on the inheritance mechanisms of C++. `BEE++` has a rich object model that separates event generation and event interpretation. The main drawback of `BEE++` is that the instrumentation of the application has to be performed by the programmer manually.

`ATOM` [14] is a framework for the construction of program analysis tools. It takes a set of object files, a file containing the instrumentation routines, and a file containing the analysis routines. Instrumentation is started by invoking the `Instrument` procedure written by the user. This procedure allows the selection of the routines contained in the object code. Similar to AOP techniques it is possible to

insert calls before and after certain points within the object code. Within ATOM these points can be procedures or even assembly instructions. ATOM, however, has some drawbacks concerning program comprehension. First, the structure of the source code is no longer visible within the instrumentation routine since it works on the object file level. Second, inserting instrumentation code for different procedures starts to become difficult if the parameters stored within registers shall be considered.

Form [13] can be used to construct tools for analyzing the runtime behavior of standalone and distributed software systems. Form is somehow similar to ARE since it collects data during an application run and visualizes call graphs with UML sequence diagrams. While Form avoids the instrumentation of the applications and uses the Java Virtual Machine Profiling Interface (JVMPPI) to collect data ARE uses AspectJ to instrument them. Since Form uses the Java Virtual Machine's profiling interface, it can analyze programs where source code is unavailable. A disadvantage of this approach, however, is that the profiling interface is tuned for the performance analysis and hence few notifications such as the the invocation of methods or creation and termination of threads can be used for program analysis. Unlike with our approach, information such as the parameters passed cannot be obtained as, for instance, is necessary for the analysis of reflective method calls.

## 8. Conclusions

The contribution of this paper is the demonstration of how aspect-oriented programming can help to instrument software applications to be reverse engineered. As we have shown, aspect-oriented programming, in particular AspectJ, allows us to gather both dynamic execution and runtime data of the running software application.

The system we have implemented consists of a support library, that provides a series of callbacks for the collection and analysis of trace-data. Our callbacks are woven automatically into the application to be reverse engineered, hence it is not necessary to understand the underlying software application.

ARE, our reverse engineering tool, allows us to filter the data collected with respect to different interests. Providing a powerful filtering mechanism as shown in Section 4 is mandatory since otherwise the data to be analyzed would be overwhelming. Using our support library, it is possible to limit the data collection to specific classes or even to specific instances thus enabling a focused analysis of the underlying application. Additionally, by having access to execution data such as the arguments of a method call, we can analyze reflective (dynamic) method calls.

To demonstrate the feasibility of our approach, we have reverse-engineered, the Bean Builder development environ-

ment from Sun Microsystems and have extended it with the functionality to display a user-defined menu and to record and playback macros. As we have shown in Section 5 the application has been extended on the basis of the data gathered using ARE. Both of the extensions have been added using AspectJ, hence again requiring no changes to the original code. This example also demonstrates that our approach is compatible with applications that themselves were developed with AspectJ.

Although, our approach does not require the engineer to read and understand an application's source code, it requires the availability of the application's source code. As we have shown, however, our approach only requires the application's source code for AspectJ being able to weave in the necessary instrumentation. Thus, it would immediately profit from the availability of an aspect compiler operating directly on Java byte-code. According to [8], it is likely that this support will be added in AspectJ 1.1.

## Acknowledgments

Thanks to Martin Pinzger for our discussions on early draft versions of this paper and to Mehdi Jazayeri for his continuous support of this project.

This project was supported in part by an IBM University Partnership Award from IBM Research Division, Zurich Research Laboratories and as part of the EASY-COMP project (IST-1999-14191) by the European Union.

## References

- [1] Aspect-Oriented Software Development Website. <http://www.aosd.net/>.
- [2] U. Aßmann and A. Ludwig. Introducing Connections into Classes with Static Metaprogramming. In P. Ciancarini and A. Wolf, editors, *3rd Int. Conf. on Coordination*, number 1594 in LNCS. Springer, Apr. 1999.
- [3] B. Bruegge, T. Gottschalk, and B. Luo. A framework for dynamic program analyzers. In *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, pages 65–82. ACM Press, 1993.
- [4] T. Eisenbarth, R. Koschke, and D. Simon. Aiding Program Comprehension by Static and Dynamic Feature Analysis. In *Proceedings of the International Conference on Software Maintenance*. IEEE, Nov. 2001.
- [5] M. Fowler and K. Scott. *UML Distilled*. Addison-Wesley, June 1998.
- [6] B. Joy, G. Steele, J. Gosling, and G. Bracha. *The Java Language Specification*. Addison-Wesley, 2nd edition, June 2000.
- [7] G. Kiczales. Beyond the Black Box: Open Implementation. *IEEE Software*, 13(1):8–11, Jan. 1996.

- [8] G. Kiczales. [Aspectj] 1.1 Plans. AspectJ Mailing List, July 2002. <http://aspectj.org/pipermail/users/2002/011664.html>.
- [9] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In M. Akşit and S. Matsuoka, editors, *Proceedings of the 11th European Conference on Object Oriented Programming (ECOOP'97)*, pages 220–242. Springer-Verlag, 1997.
- [10] R. Koschke. *Atomic Architectural Component Recovery for Program Understanding and Evolution*. PhD thesis, Universität Stuttgart, 2000.
- [11] D. Mahrenholz, O. Spinczyk, and Wolfgang-Schröder-Preikschat. Program Instrumentation for Debugging and Monitoring with AspectC++. In *Proceedings of the 5th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*. IEEE, 2002.
- [12] H. Ossher and P. Tarr. Hyper/J: multi-dimensional separation of concerns for Java. In *Proceedings of the 23rd international conference on Software engineering*, pages 821–822. IEEE Computer Society, 2001.
- [13] T. S. Souder, S. Mancoridis, and M. Salah. Form: A Framework for Creating Views of Program Executions. In *Proceedings of the International Conference on Software Maintenance*. IEEE, Nov. 2001.
- [14] A. Srivastava and A. Eustace. ATOM: a system for building customized program analysis tools. In *Proceedings of the ACM SIGPLAN '94 conference on Programming language design and implementation*, pages 196–205. ACM Press, 1994.
- [15] Sun Microsystems. *Java Platform Debugger Architecture*, 1.4 edition. <http://java.sun.com/j2se/1.4/docs/guide/jpda/>.
- [16] Sun Microsystems. *Java Core Reflection: API and Specification*, Feb. 1997. <http://java.sun.com/products/jdk/1.1/docs/guide/reflection/>.
- [17] Sun Microsystems. The bean builder 1.0 beta. <http://java.sun.com/products/javabeans/beanbuilder/>, Jan. 2002.
- [18] N. Wilde and C. Casey. Early Field Experience with the Software Reconnaissance Technique for Program Comprehension. In *Proceedings of the International Conference on Software Maintenance*. IEEE, 1996.