Retrieving Sequence Diagrams from Aspect-Oriented Systems

CS842: Aspect-Oriented Programming Course Project

Steven She (shshe@uwaterloo.ca)

Abstract

Aspect-oriented programming is built on the concept of separating concerns. While separation of concerns reduces textual scattering and tangling by encapsulating concerns within a localised module, the behaviour of an aspect-oriented program becomes scattered. Capturing the sequential behaviour of an aspect-oriented program is essential for the validation of the program's run-time semantics.

SequenceRetriever, a tool for retrieving UML sequence diagrams during the execution of a program is presented. The *SequenceRetriever* tool is built on top of an extensible framework which facilitates the development of additional trace components and diagram types. An AspectJ trace component and an Eclipse TPTP trace component is implemented. Using the two trace components, sequence diagrams of several programs is presented. A comparison between the AspectJ and TPTP sequence diagrams reveal the ajc weaver implementation of several aspect-oriented constructs.

Contents

1 Introduction

Documentation and source code are invariably linked together, however, existing techniques rely on manually propagating changes from source code to documentation. Diagrams embedded in the documentation are often manually created and updated in order to accurately reflect the implementation of a system. Behavioural diagrams are often written to describe the expected behaviour of a system. In particular, sequence diagrams are used to describe interaction between object instances in a system.

Aspect-oriented programming (AOP) [11] introduces new abstraction constructs to conventional object-oriented programming (OOP) techniques. Aspects allow crosscutting concerns to be encapsulated textually in one location. While a concern is textually local in AOP, the behaviour of a program becomes scattered in multiple classes and aspects. Errors in pointcut descriptors can have widespread consequences on a program. Understanding the behaviour of an aspect-oriented system is critical to the validation of it's run-time semantics.

In this paper, *SequenceRetriever*, an extensible framework for sequence diagrams from an aspect-oriented program is presented. Advice, such as *before*, *after*, and *around* advice are captured from the execution trace by *SequenceRetriever*. Execution information is gathered through the use of two different tools. One using AOP, using AspectJ, and the other, using code instrumentation with the tools in the Eclipse TPTP project.

In section 2, the technologies and tools used to implement *SequenceRetriever* are presented. *SequenceRetriever* is introduced in section 3. In section 4, three example traces are presented. Section 5 describes related work. Section 6 describes future extensions to the tool, and section 7 concludes.

2 Tools and Technologies Used

Several tools and technologies are used by *SequenceRetriever* in order to capture program events and generate diagrams. AspectJ and the Eclipse Test and Performance Tools Platform (TPTP) project are used to capture execution events necessary to create the diagram. The UML is used in order to represent the sequence diagram. Each of these technologies and their applicable features are described in the following section.

2.1 AspectJ

AspectJ [10] is an aspect-oriented extension to the Java programming language. Traditionally, AspectJ performed weaving during program compilation. In order to weave or remove an aspect

into a program, the program needed to be recompiled. With the release of AspectJ 5, Load-Time Weaving (LTW) was introduced. LTW defers the weaving process until a class is loaded by the Java class loader at run-time. The weaving process is the same and the bytecode generated by LTW is identical to using compile-time weaving. As a result, the code executed using LTW is no different than the code executed using compile-time weaving. Load-time weaving is capable of weaving aspects directly into bytecode, without the need for source code.

AspectJ's load-time weaver is configured using an XML configuration file. The contents of the configuration file used for the sequence diagram retrieval is shown in figure 1. The `aspects` element declares which external aspects are to be woven by the load-time weaver. The `weaver` element specifies which classes in the base program are to be woven into.

```
<?xml version="1.0" encoding="UTF-8"?>
<aspectj>
    <aspects>
        <aspect name="reverseuml.aspect.ReverseSequence"/>
    </aspects>
    <weaver options="-verbose">
        <include within="reverseuml.examples..*"/>
    </weaver>
</aspectj>
```

Figure 1: `aop.xml` Load-Time Weaving Configuration File

SequenceRetriever makes use of AspectJ's LTW feature. Since the code instrumentation and diagram retrieval is an ancillary concern to the base program, it is beneficial to be able to selectively weave the *SequenceRetriever* aspect at deployment. However, LTW is optional, and the *SequenceRetriever* aspect can also be weaved in at compile-time.

2.2 Eclipse Test and Performance Tools Platform

The Eclipse TPTP project [7] is a framework currently developed by the Eclipse foundation that provides testing and profiling tools to the Eclipse IDE. The Tracing and Profiling project in TPTP was of particular interest as it provided a framework for collecting, analysing and augmenting a program's execution.

TPTP collects information from the Java Virtual Machine (JVM) through a separate agent process. An agent is attached to a JVM and provides instrumentation data to an agent controller. The agent uses the Sun JVM profiling interface (JVMPI in Java 1.4 and JVMTI in Java 5) in order to instrument the executing code.

TPTP contains a feature called *probekits*, which inserts Java code fragments into specified locations in the execution path of a program. Table 1 provides a summary of the various join points that are exposed to probekits.

Probekit	AspectJ Join Point
entry	**before execution**(..)
exit	**after execution**(..)
before call	**before call**(* `<Targets>`(..))
after call	**after call**(* `<Targets>`(..))
caught exception	**before handler**(Exception+) && **args**(e)
class static initialiser	**before staticinitialization**(`<Targets>`)
executable unit	**before call**(* *(..)) && **within**(`<Targets>`)

Table 1: Mapping from Probekit capabilities to AspectJ terminology

TPTP also exposes contextual information to the probe fragments. The class name, method name, method signature, this object, exception object, method arguments and returned object are exposed. These data elements are also present as a pointcut such as this or target in AspectJ, or as an attribute of the thisJoinPoint object.

2.3 UML 2.0

UML 2.0 is a standard for describing system interactions [5] put forth by the Object Management Group (OMG). UML 2.0 contain a set of diagrams that share a common meta model. The semantics of UML 2.0 are formalised using the Meta-Object Facility (MOF) meta model. The underlying MOF meta model provides UML 2.0 with a standard interchange format, XMI, and a set of operations for model transformation.

The output model of the *SequenceRetriever* is a UML sequence diagram. The sequence diagram shows the message interaction between object instances during the execution of a system. As a result, a sequence diagram can been seen as a visualisation of a program trace for a given test case.

3 *SequenceRetriever*

SequenceRetriever is a framework written in Java that retrieves a sequence diagram during the dynamic execution of a program. *SequenceRetriever* provides an extensible framework for capturing, analysing and creating diagrams from the execution of a program. Currently, *SequenceRetriever* is capable of capturing object instantiation, method calls between objects, static method calls, advice invocation from aspects and creating sequence diagrams.

3.1 UML Sequence Diagram

SequenceRetriever currently support the following UML 2.0 sequence diagram elements:

- Lifelines / Instances
- Synchronous messages
- Creation messages
- Execution occurrences

A lifeline represents an instance of an object in the sequence diagram. Synchronous messages and creation messages are captured by *SequenceRetriever*. Static method calls are captured as method calls to their static class objects. Execution occurrences represent the period of time in which an instance is active during the execution of a program. These elements are automatically created when a message is received by an instance or when an object is created. Execution occurrences terminate when a reply message is sent.

3.2 *SequenceRetriever* Framework

The *SequenceRetriever* framework was created in order to capture the events run-time events of a program, interpret the events and generate diagram elements. A UML class diagram of the framework is presented in Figure 2.

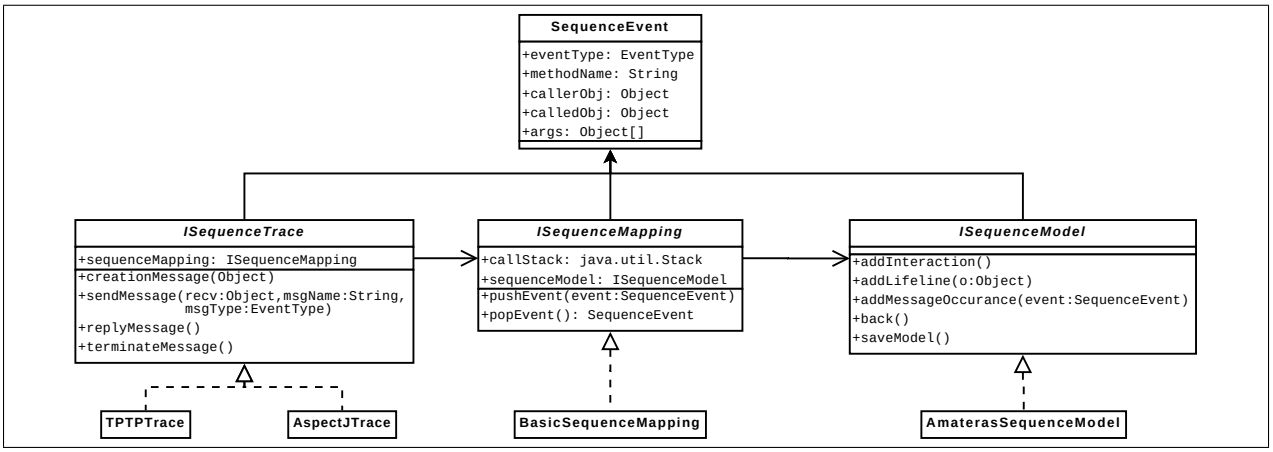

Figure 2: *SequenceRetriever* framework

The *SequenceRetriever* framework adheres to a pipe-and-filter architecture. The *SequenceRetriever* framework consists of three components: Trace, Mapping, and Model Builder. The Trace component is responsible for the retrieval of trace data from the program execution. The Mapping component translates trace data into sequence model elements described in section 3.1. The Model Builder component creates the diagram using a sequence diagram implementation. These components are described in further detail in the following sub-section.

3.2.1 Trace Component

The Trace component captures the run-time events of a program. Two trace components were implemented, one using AspectJ and another using the Eclipse TPTP framework. These trace components are described in further detail below.

```
pointcut creation() : call(*..new(..))
    && !within(ReverseSequence);
pointcut object_calls(Object caller, Object callee) : call(* *(..))
    && this(caller) && target(callee)
    && !call(static * *(..)) && !within(ReverseSequence);
pointcut static_calls(Object o) : call(* *(..)) && target(o)
    && withincode(static * *(..)) && !within(ReverseSequence);
pointcut within_static_calls() : call(static * *(..))
    && withincode(static * *(..)) && !within(ReverseSequence);
pointcut advice_calls(Object o) : call(* *(..)) && target(o)
    && adviceexecution() && !within(ReverseSequence);
pointcut advice_execution() : adviceexecution() && !within(ReverseSequence);
```

Figure 3: Pointcuts for the AspectJ trace component

AspectJ Trace Component The AspectJ trace component uses AspectJ to capture the run-time events of interest in retrieving the sequence diagram. Several pointcuts, shown in figure 3, were required to capture the necessary calls to create the sequence diagram.

The creation() pointcut is used to capture object creation. The object_calls(..) pointcut is used to capture calls between objects. Due to the use of this(..) and target(..) pointcuts, calls to and from static methods are not captured.

Static calls require a different set of pointcuts from object-to-object calls since the this object does not exist. In the sequence diagram, a static call is represented as a call to the Class object in which the static operation resides. In AspectJ, the static class is the Class object returned by the method thisJoinPoint.getSignature().getDeclaringType(). Two pointcuts are necessary to instrument static calls. The static_calls(..) pointcut captures calls to objects from a static method, and the within_static_calls() pointcut captures calls where both caller and callee are static methods.

Advice execution is captured using two pointcuts, advice_calls(..) and advice_execution(). The advice_calls(..) pointcut captures calls from within advice executions. The advice_execution() creates a behaviour execution for advice execution in the sequence diagram.

A simple class name filter was implemented in the AspectJ Trace component in order to select methods and classes of interest to the diagram. The filter is a simple boolean predicate on class and method names. Currently, calls to java.io.PrintStream and java.lang.* can be ignored by the AspectJ trace component.

Currently, AspectJ does not weave aspect code into classes in the Java library. As a result, callbacks from the Java library are not captured by the AspectJ Trace component.

TPTP Trace Component The Eclipse TPTP trace component uses a probekit, from the TPTP framework to capture the events in order to create the sequence diagram. The sequence diagram is created by instrumenting method entry and method exit join points.

The TPTP Trace Component captures the execution join point in order to construct the sequence diagram. The AspectJ Trace Component captures call join points in order to construct the sequence diagram. For sequential non-AOP programs, the sequence diagram generated by TPTP and AspectJ Trace components should be identical, however, parallel programs may retrieve sequence diagrams with minor differences between the two trace components. For aspect-oriented programs however, TPTP will capture the additional woven calls since it instruments code at the JVM level.

The probekit intercepts all system and user calls in a program by default. A targets parameter is provided to restrict the scope of the probekit. However, unlike the weaver parameter used by the AspectJ load-time weaver, the targets parameter is based on the callee (ie. target object), whereas the AspectJ weaver parameter is based on calling object (ie. this object). As a result, it is difficult to capture all calls in program, since library classes would need to be explicitly included in the probekit targets parameter.

TPTP does not provide the class object when static methods are executed. In order to capture these calls in the sequence diagram, Java's reflective interface was used to retrieve the the class based on the method signature of the executed operation.

3.2.2 Mapping Component

The Mapping component consist of two methods: pushEvent(Event), and popEvent(Event). The role of the Mapping component is to implement a layer of reasoning to the *SequenceRetriever* framework. Currently, a single Mapping component is implemented, the Basic Sequence Mapping.

The Basic Sequence Mapping passes events from the Trace component to the Model Builder component. A single stack is maintained in order to capture the call stack of single threaded applications. A simple pattern matcher was implemented in order to add rules for describing the start and finish state of the retrieved sequence diagram. Currently, method names can be specified as the trigger methods. The start and end patterns are read from the mapping.properties file which must be located in the class path.

3.2.3 Model Builder Component

The Model Builder component is responsible for constructing the sequence diagram. The methods contained in the Model Builder correspond to the sequence diagram elements.

Currently, a single model builder was implemented, the Amateras Model Builder. The Amateras Model Builder component uses the AmaterasUML Sequence Diagram API [2] (AmaterasUML API) in order to construct sequence diagrams. The diagrams created by the AmaterasUML API are stored in a custom XML format and not in the XMI format necessary for UML2 compliance. An XMI importer and exporter component is being developed for AmaterasUML, however, it is incomplete at the time of writing.

An EMF UML2 Model Builder component is under development. This is further described in future work, section 6.

4 Example Sequence Diagrams

In this section, three example programs and their *SequenceRetriever* sequence diagrams are presented. Sequence diagrams are retrieved using both AspectJ and TPTP Trace components. The diagrams were outputted to the AmaterasUML format. Creation messages currently appear as standard call messages with the name "[Creation]". Lifelines that are prefixed by <s> refer to the static class objects.

4.1 The Simple Aspect-Oriented Program

The Simple Aspect-Oriented Program (SAOP) is a small example demonstrating the capabilities of the *SequenceRetriever* tool. The base program of the SAOP performs an object instantiation, a polymorphic method call, followed by a call that throws an exception. An aspect which executes *before*, *after*, *after returning*, and *after throwing* advice is present. The standard output of the SAOP is shown in figure 5.

The sequence diagram retrieved using the AspectJ *SequenceRetriever* is shown in figure 6. In figure 7, the sequence diagram retrieved using the TPTP *SequenceRetriever* is shown.

The sequence diagram retrieved by TPTP contains more calls than it's AspectJ counterpart. The events are retrieved directly from the JVM. As a result, the additional method calls and hooks added by the AspectJ compiler during the weaving process are visible. In the case of the SAOP, a hook is added by `ajc` to the static class initialiser in order to create the aspect object, `LoggingAspect`. The additional calls to the `aspectOf()` method are also not present in the AspectJ version.

```
public class LoggingMain {
	public static void main(String[] args) {
		System.out.println("Main Starting.");
		Animal myDog = new Dog();
		String a = myDog.sound();
		try {
			myDog.throwsException();
		}
		catch (Exception e) {
			System.out.println("Exception caught.");
		}
		System.out.println("Main Finished.");
	}
}

public aspect LoggingAspect {
	pointcut calls() : call(* reverseuml.examples.aspects..*(..))
		&& !within(LoggingAspect);

	before() : calls() {
		System.out.println("Aspect: Before call to "
				+ thisJoinPoint.getSignature() + "...");
	}
	String around() :
			call(String reverseuml.examples.aspects..sound(..)) {
		return proceed();
	}
	after() : calls() {
		System.out.println("Aspect: After call to "
				+ thisJoinPoint.getSignature() + "...");
	}
	after() returning (String s):
			call(String reverseuml.examples.aspects..*(..)) {
		System.out.println("Aspect: After Returning... " + s);
	}
	after() throwing (Exception e) : calls() {
		System.out.println("Aspect: After Throwing...");
	}
}
```

Figure 4: Simple Aspect-Oriented Program Example

```
Main Starting.
Aspect: Before call to String reverseuml.examples.aspects.logging.Animal.sound()...
Aspect: Around call...
Dog Sound "bark"
Aspect: After Around call...
Aspect: After call to String reverseuml.examples.aspects.logging.Animal.sound()...
Aspect: After Returning... "bark"
Aspect: Before call to void reverseuml.examples.aspects.logging.Animal.throwsException()...
Aspect: After call to void reverseuml.examples.aspects.logging.Animal.throwsException()...
Aspect: After Throwing...
Exception caught.

Main Finished.
```

Figure 5: Output from the Simple Aspect-Oriented Program

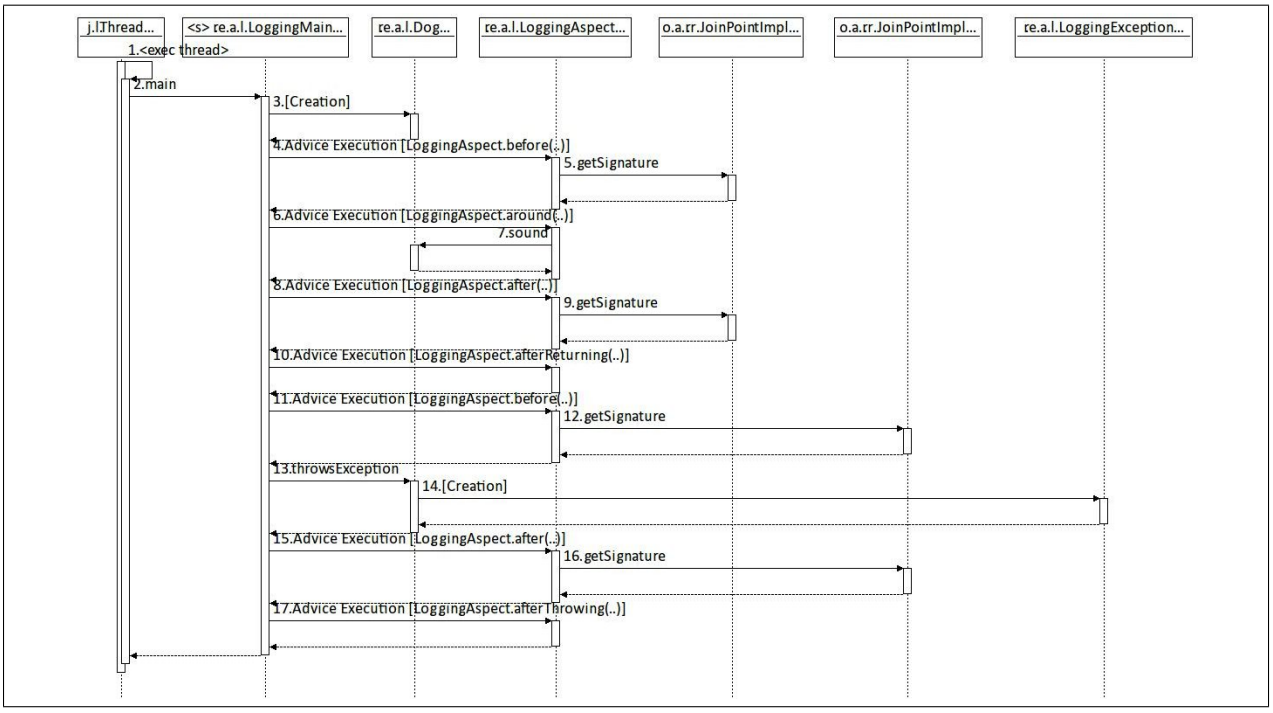

Figure 6: Simple program trace using AspectJ Trace Component

A notable difference from the AspectJ sequence diagram is the TPTP version does contain a call into the LoggingAspect object when invoking around advice. This behaviour is due an optimisation performed by ajc, which creates specialised methods for around advice at the join point shadows [9]. In the sequence diagram, the in-lined (specialised) method calls are visible as messages 9 and 10. An sequence diagram depicting closures for *around* advice is described in sub-section 4.2.

The AspectJ sequence diagram is able to capture the aspect semantics better than the TPTP sequence diagram. This is not surprising, since the SAOP was written using AspectJ. All types of advice invocation appear uniformly in the AspectJ sequence diagram. In particular, *around* advice appears as a message to the LoggingAspect instance. While this behaviour matches with our intuitive understanding of aspects, it does not match with the underlying implementation of around advice. As mentioned previously, ajc implements *around* advice as either an in-lined (specialised) method, or as a closure While the AspectJ version of *SequenceRetriever* is able to properly capture the semantics of AspectJ programs, it is not applicable to other AOP languages. A generalised method of capturing aspect-oriented semantics is described further in section 6.

4.2 AspectJ Closure

When weaving *around* advice, the AspectJ compiler, ajc, generates closures when encountering proceed() statements in anonymous classes. In figure 8, a nonsensical closure example is shown. In this program, the execute(int) call is advised by *around* advice. The *around* advice instantiates an anonymous class which contains a call to proceed(). As a result, a closure is generated by the ajc compiler during weaving. Executing this program with the TPTP *SequenceRetriever* creates the sequence diagram shown in Figure 8.

The ajc closure implementation of *around* advice creates an additional private class, TestClass$ajcClosure1. The lifeline ClosureAspect$1 refers to the anonymous class instantiated from AnonymousInterface. Starting at message 8, the woven closure code begins execution. The closure object is created, and the *around* advice is invoked at message 9. Message 10 corresponds to the anonymous class instantiation. At message 12, a call to a static method of ClosureAspect is made. The closure object is executed at message 13 and at message 14, the *proceed* statement is invoked resulting in the method TestClass.execute(int) being called.

In figure 10, the AspectJ sequence diagram is shown. By using the adviceexecution() pointcut in AspectJ to capture advice invocation, the ajc closure implementation is not exposed. Similar to the in-lined *around* advice implementation shown for the SAOP, the AspectJ sequence diagram depicts the *around* advice as a message to the ClosureAspect instance.

Figure 7: Simple program trace using TPTP Trace Component

```
public class ClosureMain {
        public static void main(String[] args) {
                TestClass testClass = new TestClass();
                System.out.println(testClass.execute(0));
        }
}
public aspect ClosureAspect {
        public interface AnonymousInterface {
                public String getNumber(int i);
        }
        String around(int numOfTimes) : execution(String *..execute(int))
                        && args(numOfTimes) {
                System.out.println("Inside closure aspect");
                return new AnonymousInterface() {
                        public String getNumber(int i) {
                                //Method which prefixes the return value with i
                                return "" + i + proceed(i);
                        }
                }.getNumber(numOfTimes);
        }
}
```

Figure 8: Closure Example

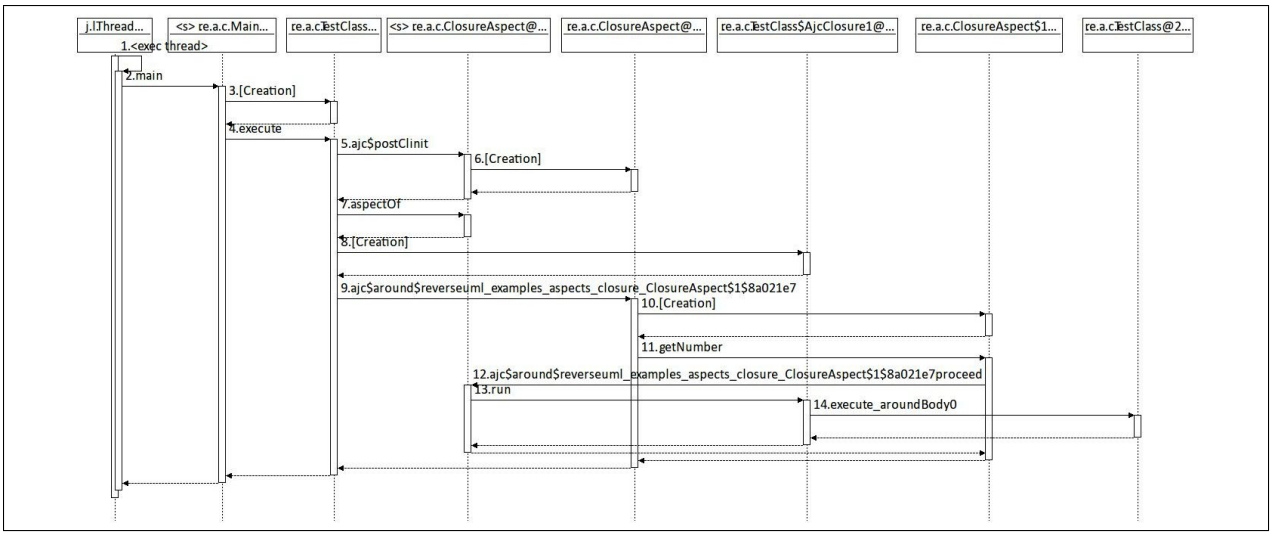

Figure 9: TPTP Sequence Diagram of the Closure Example

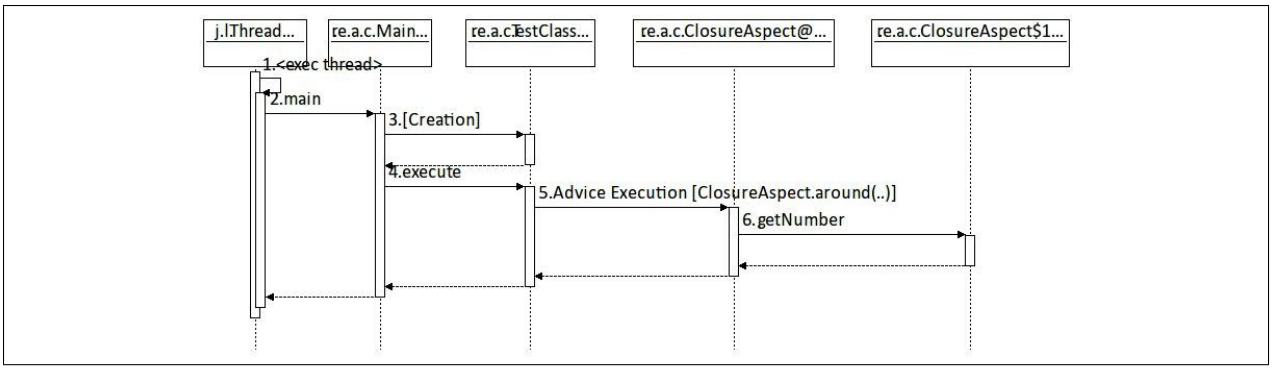

Figure 10: AspectJ Sequence Diagram of the Closure Example

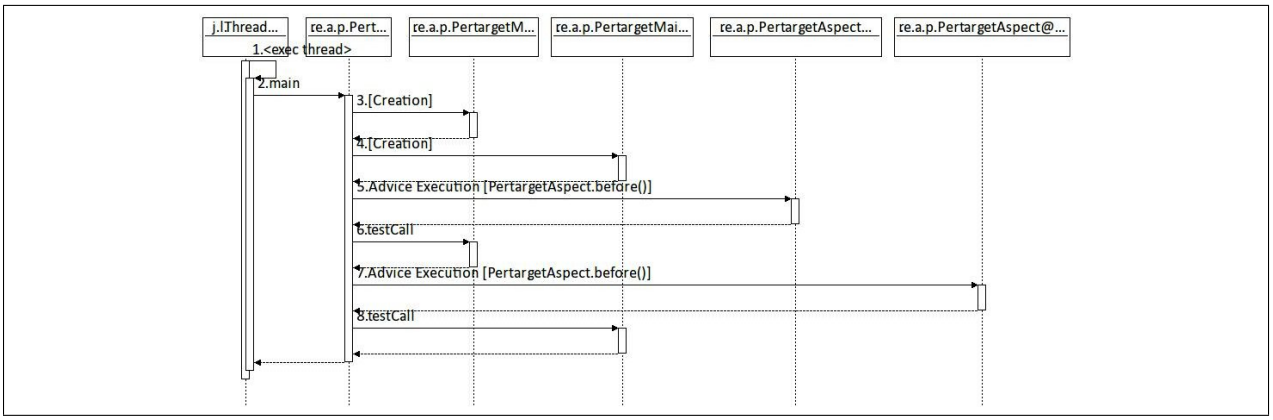

Figure 11: AspectJ Sequence Diagram of the pertarget Example

4.3 Pertarget

The pertarget(Pointcut) aspect creates a new aspect object for every target object that is selected by the pointcut. In figures 11 and 12, the sequence diagrams of a simple pertarget example is shown. In the example, the call to testCall() is advised by a pertarget aspect. Two PertargetMain objects are created at the start of the program. (messages 3 and 4). In the TPTP sequence diagram, we notice that the aspect weaver creates the pertarget aspects when the join point is reached. A reference to the pertarget aspect in the target object (PertargetMain) is set (messages 8 and 18 in the TPTP trace) and later used to retrieve the pertarget aspect object.

5 Related Work

Briand *et al.* [3] present a method of extracting UML sequence diagrams through a combination of static and dynamic analysis. Static analysis is used to gather the conditions from if statements and

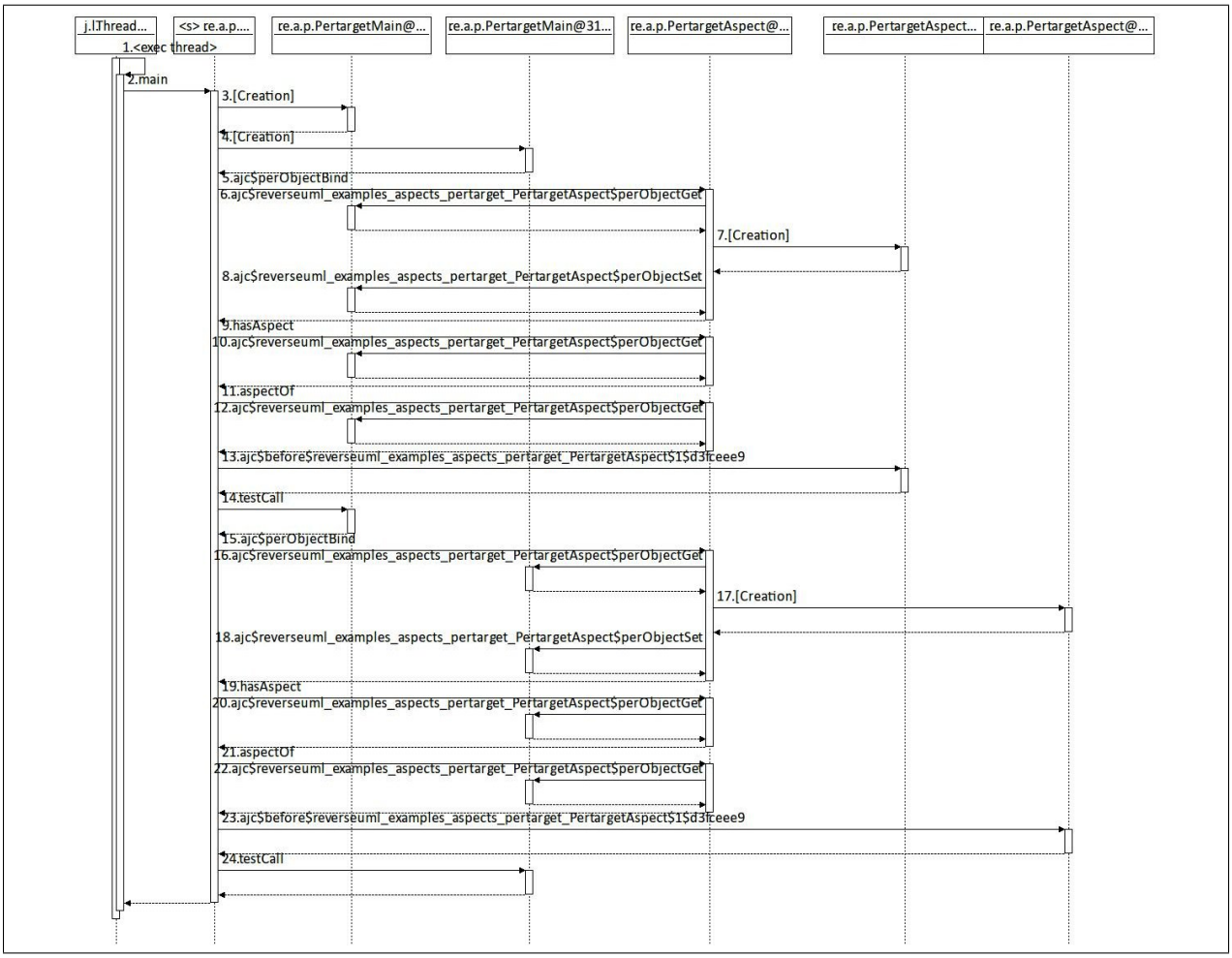

Figure 12: TPTP Sequence Diagram of the pertarget Example

loops. Their method targets C++ code and uses an instrumentation tool written in Perl. Similar to approach taken by *SequenceRetriever*, Briand *et al.* map trace events to a higher level meta model.

There has also been work in the static reverse engineering of sequence diagrams. Rountev *et al.* [12] describe a method of using control-flow analysis to create sequence diagrams. A benefit of using static analysis is that control structures are clearly visible in the source code, and as a result, accurate UML combined fragments can be created. Aspect-oriented concepts are not covered by the authors.

Guéhéneuc and Ziadi [8] present a position paper on the automated reverse engineering UML 2.0 sequence diagrams. They propose to merge different program executions to form UML combined fragments that describe loops and conditionals. They also propose to use static analysis in combination with dynamic analysis techniques in order to retrieve control structures from source code.

6 Future Work

EMF UML2 Meta Model A short term goal for *SequenceRetriever* is to implement the EMF UML2 Meta Model [6] from the Eclipse Model Development Tools (MDT) project as a Model Builder component. EMF UML2 is formalisation of the UML2 meta model using the Eclipse Modelling Framework (EMF). The Eclipse Graphical Modelling Framework (GMF) project provides editing and display capabilities to EMF UML2 models. However, absent from the current GMF release is the support for sequence diagrams. In addition, the EMF UML2 is a very heavyweight model as it consists of the entire UML2 specification and all of its diagram types. As a result, *SequenceRetriever* was first developed with the AmaterasUML model, however, creating EMF UML2 models from *SequenceRetriever* would be a necessary step for future work.

UML2 Combined Fragments Extending the framework for UML2 combined fragments is future work. The Event interface would need to be extended in order to capture loops and conditional statements. In order to capture combined fragments, a *merge* operator for multiple sequence diagrams would need to be implemented. For aspect-oriented systems, UML2 combined fragments could be created for pointcut definitions. Pointcuts are analogous to conditions on the current state of execution. If the source code was not available, it may be possible to reverse engineer pointcut definitions from combining multiple traces of an aspect-oriented program.

Sequence Diagram Verification Future work includes the verification of the retrieved sequence diagram against a human-specified sequence diagram. During the design stage of development, sequence diagrams are often written to describe the expected behaviour of a system. Comparing

design-time sequence diagrams to the reverse engineered sequence diagrams would be analogous to validating the results of a test case to the specifications of the designer. However, the mapping between design-time sequence diagram elements and retrieved elements is not straightforward. For example, a particular call may be described as a single message in the design-time sequence diagram, however, in the implementation, it may be a set of calls, with callbacks in between. The problem of comparing diagrams is similar to the problems of model merging. Brunet *et al.* [4] describe model merging as a algebraic operator over models and relationships. Operators such as *match*, *diff*, and *check-property* are described in the paper. These operators would be greatly applicable to the problem of comparing sequence diagrams.

Extending Match and Filtering Criteria The existing filtering and matching criteria in *SequenceRetriever* is primitive. Using tracematches as a Trace component will allow for more powerful trace queries as well as a more efficient implementation. Tracematches [1] provide a means of executing advice based on the history of computation. The execution trace is filtered in terms of a set of symbols. For sequence diagrams, a tracematch can be viewed as a method of specifying a sequence of interested events.

Inference Engine Using the framework built for *SequenceRetriever*, an improved Mapping component could be built. A rule engine can also be implemented in the Mapping component in order to infer abstract diagram elements from trace information. One particular use is to facilitate the creation of other diagram types. System artefacts can be reconstructed based on inference rules and heuristics. This approach would be similar to the DiscoTect system by Yan *et al.* [13]. For aspect-oriented systems, implementation patterns, such as the closure pattern from `ajc` can be encoded as a rule in the inference engine.

7 Conclusions

Understanding the behaviour of a program is essential to the validation of it's run-time semantics. With the introduction of new abstraction techniques, such as AOP, the gap between the textual representation of a program and it's behavioural semantics begin to widen. In AOP, aspects enable programmers to encapsulate crosscutting concerns into a module. While, encapsulation localises a concern's source code in one location, the code can still affect multiple points of execution. In addition to aspects, conventional OOP techniques add polymorphism and inheritance to further separate code from behaviour.

In this paper, *SequenceRetriever*, an extensible framework for retrieving sequence diagrams during a program execution is presented. *SequenceRetriever* is able to instrument executing code through

the use of AspectJ and the Eclipse TPTP framework. *SequenceRetriever* is capable of capturing and displaying advice invocation. Using the AspectJ trace component of *SequenceRetriever*, advice semantics are better captured in the retrieved diagram. However, the advice semantics of the AspectJ trace component cannot be generalised to other aspect-oriented languages. Using the TPTP trace component, a lower level sequence diagram is retrieved, however, the information contained in the diagram can be used as a method of examining woven code.

Retrieving diagrams using dynamic analysis provides an accurate representation of the behaviour of the system. Extending the *SequenceRetriever* framework with new mapping components can create new rules for determining the creation of diagram elements. Creating new model builder components can create new diagram types. *SequenceRetriever* can be used as a platform to examine the behaviour of systems.

The reverse engineering of sequence diagrams provide a method of retrieving object interactions from the execution of a system. *SequenceRetriever* is a framework which provides an extensible means of retrieving trace information and subsequently generating diagram elements. It is the goal of *SequenceRetriever* to help close the gap between code and documentation.

References

[1] C. Allan, P. Avgustinov, A. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding trace matching with free variables to AspectJ. In *Proceedings of the 20th annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 345–364, 2005.

[2] Project Amateras. http://amateras.sourceforge.jp/. AmaterasUML.

[3] L. C. Briand, Y. Labiche, and Y. Miao. Towards the reverse engineering of uml sequence diagrams. In *WCRE '03: Proceedings of the 10th Working Conference on Reverse Engineering*, page 57, Washington, DC, USA, 2003. IEEE Computer Society.

[4] Greg Brunet, Marsha Chechik, Steve Easterbrook, Shiva Nejati, Nan Niu, and Mehrdad Sabetzadeh. A manifesto for model merging. In *GaMMa '06: Proceedings of the 2006 international workshop on Global integrated model management*, pages 5–12, New York, NY, USA, 2006. ACM Press.

[5] Hans-Erik Eriksson, Magnus Penker, and David Fado. *UML 2 Toolkit*. John Wiley & Sons, Inc., New York, NY, USA, 2003.

[6] The Eclipse Foundation. http://www.eclipse.org/modeling/mdt/. Model Development Tools (MDT).

[7] The Eclipse Foundation. http://www.eclipse.org/tptp/. Eclipse Test & Performance Tools Platform Project.

[8] Yann-Gael Guéhéneuc and Twefik Ziadi. Automated reverse-engineering of uml 2.0 dynamic models, 2006.

[9] E. Hilsdale and J. Hugunin. Advice weaving in AspectJ. In *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development*, pages 26–35, 2004.

[10] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. Springer–Verlag, 2001.

[11] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object–Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, 1997.

[12] Atanas Rountev, Olga Volgin, and Miriam Reddoch. Static control-flow analysis for reverse engineering of uml sequence diagrams. In *PASTE '05: The 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 96–102, New York, NY, USA, 2005. ACM Press.

[13] Hong Yan, David Garlan, Bradley Schmerl, Jonathan Aldrich, and Rick Kazman. Discotect: A system for discovering architectures from running systems. In *Proceedings of the 26th International Conference on Software Engineering*, Edinburgh, Scotland, 23-28 May 2004.