TEAMS

The Extensible Aspect-Oriented Monitoring System

Gholamali Rahnavard
PLEASE Lab, Department of Computer Science
New Mexico State University

February 1, 2014

Contents

1 Introduction 1
2 Prerequisites 2
2.1 TEAMS 2
2.2 ASM 2
2.3 ANTLR 2
2.4 Ant 2
3 Configuration 2
4 Using TEAMS 2
4.1 Write an Aspect Program 3
4.2 Write an Advice 3
4.3 Run TEAMS 4
5 Extending TEAMS 4
5.1 Point-Cut Designator 5
5.2 Point-Cut Shadow Matcher 6
5.3 Information 9
5.4 Advice Execution Model 10

1 Introduction

The development of reliable software often requires the dynamic monitoring of the system under development for many reasons, but creating low-level instrumentation is difficult and developers are generally restricted to using what existing tools offer. It is possible to create an efficient instrumentation framework that provides an effective Aspect-Oriented Programming based high level interface for instrumentation while also providing an effective capability for extension that enables full monitoring concept coverage. The Extensible Aspect-Oriented Monitoring System (TEAMS), an aspect-oriented framework, is defined in this work to specify required instrumentation in a high-level formalism. The purposes for this framework are simplicity, extensibility, portability, and for monitoring concept coverage. The TEAMS aim is to provide researchers an easy-to-use platform for building instrumentation that will support their monitoring and analysis research, and will provide practioners the ability to craft their own analyses without needing to understand low-level instrumentation.

2 Prerequisites

2.1 Teams

Download the last version of TEAMS from `http://www.cs.nmsu.edu/please/projects/teams/` .

2.2 Asm

ASM is an all purpose Java bytecode manipulation and analysis framework. Please visit `http://asm.ow2.org/index.html` for more information. The current release of TEAMS uses asm version 5.0.

2.3 Antlr

ANTLR (ANother Tool for Language Recognition) is a powerful parser generator for reading, processing, executing, or translating structured text or binary files. Please visit `http://www.antlr.org/` for more information. The current release of TEAMS uses antlr version 4.0.

2.4 Ant

Ant is a Java library and command-line tool whose mission is to drive processes described in build files as targets and extension points dependent upon each other. The main known usage of Ant is the build of Java applications. Ant supplies a number of built-in tasks allowing to compile, assemble, test and run Java applications. Please visit `http://ant.apache.org/` for more information.

3 Configuration

Extract TEAMS-1.0.zip into an appropriate path you want to use it. For example, put it in /home/Your-Path/. Then add ASM and Antlr packages to your library directory if you don not have them already (here to the ../Your-Path/lib/). You should have antlr-runtime-4.0.jar and asm-all-5.0_ ALPHA.jar to the lib directory for current version of TEAMS. In the current lib directory you can also see etension.jar file which includes an extension of TEAMS and example.jar file which includes some advice classes example.

4 Using Teams

TEAMS is designed to be extensible for new point-cut designators. Here some basic PCDs are presented to see how TEAMS works.

4.1 Write an Aspect Program

TEMS has its own AOP language. To monitor an application, an AOP program in TEMS's AOP should be write to define the abstract monitoring concerns. the program should be written a text file and later it will be used as input to TEMS. In the current configuration AOP files are saved in AOP directory, but it can be anywhere, only the path of the program will be different when the commands line in the next section. To write an AOP program in TEMS the following structure should be used:

```
aspect MyAspect {
       pointcut MyPC(): PCD(agruments) (&& PCD(arguments))*
                        { (requires DataType.DataName;)* }* ;

(AdviceID AEM() : MyPC ;)*
}
```

The notation is used is similar to regular expression to explicate the structure. * is used to show the repetition of statements. PCDs are characterized existing point-cut designators in TEAMS. The present-day TEAMS's PCDs are: within, withincode, execution, call, basicblock and fieldAccess.

An example for this is:

```
aspect MyAspect {
       pointcut Execution(): within(org.h2) && execution(* get())
                             { requires Execution.methodName; } ;

Advice1 before() : Execution ;
}
```

4.2 Write an Advice

The mechanism that TEAMS uses for writing advices is having advice references in the aspect program and develop corresponded advices in plain old Java program. There some rules that should be preceded:

- The class name of advices should be the same as the Aspect name.
- The method that corresponds an Advice should be the combination of the advice execution model and the advice reference. For example, if the AEM is before and advice reference is Execution then the method name should be before_ Execution. The advice references are the same as point-cut name, for instance, in the last example Execution is the point-cut name and advice reference.
- The parameters of the method should present the data that been required in the aspect program. In the last sample aspect Execution need the method name which is executing. SO the method should accept a String as the parameter.

The advice can be in different path than teams bur it is necessary to add its path to the command. In the next section it uses the following advice:

```
public class MyAspect {
   public static long m =0;
   public static void before_Execution(final String method) {
     m++;
     System.out.println("You are in Advice for Execution of Method: "+ method + "  " + m);
  }
}
```

4.3 Run Teams

To test and run a demo of TEAMS, a simple application is provided in Application directory. The DaCapo benchmark also can be used as a source for the under monitoring application which it includes a set of open source applications with non-trivial memory loads. This run uses H2 as one of the applications. It can be downloaded from `http://www.dacapobench.org/download.html` and extract to your appropriate path.

Open a terminal then run the following commands:
change directory to your path:
>cd Your-Path
The abstract command line for monitoring application is:
>time java -javaagent:teams.jar=path-To-Your-Aspect-File
-cp ./lib/extension.jar:./lib/example.jar:Path-to-Application-and-Libraries:. The-main-Class-of-the-Apllication-and-its-input
Run the following command for montoring a simple application that provided in the example.jar file:
>time java -javaagent:teams.jar=AOP/SimpleExecutionAspect -cp lib/extension.jar:
lib/example.jar:lib/asm-all-5.0_ALPHA.jar:lib/antlr-runtime-4.0.jar:. example/UserClass

For example for Dacapo I used the following command line (you need to extract Dacapo package, here is in download directory):
>time java -javaagent:teams.jar=AOP/SimpleExecutionAspect -cp lib/extension.jar:lib/example.jar:lib/asm-all-5.0_ALPHA.jar:lib/antlr-runtime-4.0.jar:Application/dacapo-9.12-bach:Application/dacapo-9.12-bach/jar/h2-1.2.121.jar:. Harness h2

5 Extending Teams

The main goal of TEAMS is to be extensible. For this aim , three part of the framework should be extended to handle a new concern. To create a new PCD in Teams-Java, three things need to be done:

1. Add the grammar clause(s) that will match the designator's syntax;
2. Write the code that finds and indicates the matching execution points;

3. Write the code that provides unique data for the PCD; and

4. Develop an advice execution model, TEAMS supports before and after so far.

5.1 Point-Cut Designator

Our pointcut expression grammar is an Antlr grammar. We currently do not have an automated mechanism to extend the grammar, but adding a new designator is straightforward and mostly mechanistic. A production rule for a generic designator has an OR'd set of clauses, each a production rule for one specific PCD; a clause for the new PCD is needed, which is generally just one nonterminal for the specific PCD (e.g., call for our method call PCD). Then a new production rule for the new PCD is needed. This rule must embody the designator plus any arguments that it might have. For the call example, in full Antlr syntax this looks like

```
call returns[Designator value]:
  { $value = new Designator();
     $value.setName("call"); }
     'call' '('
e=methodSignaturePattern {$value.setArgument(e);}
')'
;
```

Curly braces surround embedded Java actions and are the same for all PCD rules; the syntax of the PCD is simply the literal "call", the literal parentheses, and the single PCD argument which is the nonterminal methodSignaturePattern, which matches a RegEx-style pattern describing methods to match (this is similar to AspectJ and other AOP implementations). We provide some existing nonterminals for standard AOP designator arguments, plus simple strings and numerical arguments. If a PCD needs something different, more Antlr clauses may need to be created for its arguments, but we think this will be rare. The Java actions simply create an object for Teams to use as it processes the expression.
To support a new point-cut designator, two part of the grammar should be extended, and then generate new parser using antlr and replace it with existing TEAMS's Parser (TEAMSPareser.java).

1. Add the new designator to the designator(non-terminal token). Let's assume that "loop" is the new designator you want to add, so you will extend the designator as showing in the following:

   ```
   designator returns[Designator  value]
       : {
           $value= new Designator();
         }
   ```

```
(

e=call
    {
    $value = $e.value;
    }
|
e7=loop
    {
    $value = $e7.value;
    }
|
.
.
.
```

2. Add your deginator, here "loop" as a non-terminal token. Simply you just need to copy and then past one of the exisiting desingnators and rename it to your designator, for example, for loop we will have:

```
loop returns[Designator value]
: {$value = new Designator();
   $value.setName("loop");}'loop' '('
   e=methodOrConstructorPattern
   {$value.setArgument($e.value);} ')';
```

3. Now the new Parser, Lexer, Listener, tokens, and Lexer tokens should be generate and replace in teams package. To do this, the grammar file will give to antlr as input and it generate all the necessary files. The command for this is:
   >java -jar antlr.jar TEAMS.g

5.2 Point-Cut Shadow Matcher

The creator of the new PCD must write code that embodies what that PCD is: to match program execution points where instrumentation can be attached. Ultimately we will be exploring multiple different views of a program execution (dimensions), but for now we focus on the most immediate and used view, that of the code being executed. Teams uses the ASM bytecode manipulation framework for inserting the instrumentation into the program, and so the new designator code must use some of the ASM mechanisms to discover and find the execution points that the designator matches. The code to implement a new PCD must implement the interface PCDShadowMatcher. For now, this interface is:

```
public interface PCDShadowMatcher {
```

```
   public Set<JoinPointShadow> match(MethodNode methodNode,
                                     ArgumentList args );
}
```

The new PCD must implement just one method, match. This method takes as its first argument an ASM object that represents a method in a class; thus match() is called once for each method of each class that is loaded. The second argument are the PCD arguments from the pointcut expression. Match() returns a set of bytecode intervals that match the PCD, marked by the positions of the first and last instructions in the interval. Many PCDs, such as the call PCD, might have just one instruction in each interval, but some, such as a hypothetical loop body PCD, could have a longer sequence of instructions that match. Note that a PCD creator in Teams-Java must learn and use the ASM API to implement their PCD's functionality; Teams is intended to hide such complexity from the Teams user, but needs to expose it to the Teams extender. CallShadowMatcher class in the extension package implement the match method which can be use as an example for this step. **The name of PCD shadomatcher class must start with the name of PCD.** The following shows how Your PCD shadowMatcher , as example loop, can be implemented. You can see the extension package to see the other implemented PCD shadow matchers.

```
package extension;

import java.util.HashSet;
import java.util.Iterator;
import java.util.Set;
import java.util.Vector;

import org.objectweb.asm.Opcodes;
import org.objectweb.asm.tree.AbstractInsnNode;
import org.objectweb.asm.tree.MethodNode;

import teams.Argument;
import teams.DynamicJoinPointInfo;
import teams.JoinPointInfoExp;
import teams.JoinPointShadow;
import teams.PCDShadowMatcher;
import teams.StaticJoinPointInfo;

public class LoopShadowMatcher implements PCDShadowMatcher {
    @Override
    public void makeArgumentValue(final JoinPointShadow joinPointShadow,
        final MethodNode methodNode,
        final Vector<JoinPointInfoExp> infoExpVect) {

       /* This method is used to produce the necessary general information and
```

```
         * argument to be sued in advice execution model to generate dynamic information.
         * It explained with an example of its implementation in next section.
        */
}

@Override
public Set<JoinPointShadow> match(final MethodNode methodNode,
   final Argument argument,
   final Vector<JoinPointInfoExp> joinPointInfoExpVector, final int pc) {

   // Use a set to keep the matching areas as shadow.
   Set<JoinPointShadow> resultShadows = new HashSet<JoinPointShadow>();

    // matching the loop argument with the current method
    if (methodNode.name.startsWith(argument.getExpression())) {

    //Create final the set for final the results
    resultShadows = new HashSet<JoinPointShadow>();

    // Create a shadow
    final JoinPointShadow currentShadow = new JoinPointShadow();

    // Set the shadow type
    currentShadow.setType("loop");

    // Set the shado's method node
    currentShadow.methodNode = methodNode;

    // Find the set the beginning of the shadow based on your definition
    // and ASM analysis
    currentShadow.setInstructionStartNumber(0);

    // find the end of the shadow it can be more than one point
    final Vector<Integer> ends = new Vector<Integer>();

    // Start to analyze the method instructions
    Iterator<AbstractInsnNode> insnNodes = methodNode.instructions.iterator();

    while (insnNodes.hasNext()) {
       final AbstractInsnNode insn = insnNodes.next();
          /*
           * Procees the code to find the neccssary pionts
           * for the shadow such as start point and end points
         */
     }
    // Set the end of the shadow (Vector of points)
```

```
    currentShadow.setInstructionEndNumber(ends);

    /* Process and produce the required data for this shadow
     * or the necessary argument to produce the dynamic data in advice execution phase
    */

    if (joinPointInfoExpVector.size() > 0) {
       makeArgumentValue(currentShadow, methodNode, joinPointInfoExpVector);
    }

    // Add the shadow to the list of the shadow as result
    resultShadows.add(currentShadow);
    }
 return resultShadows;
}
```

5.3 Information

Providing the data that are necessary to analyze an application is essential for TEAMS. These data are categorized in to two types: general and dynamic data. During shadow matching TEAMS the general data and the argument that are essential to produce dynamic data should be generate. There are three main places(steps) to request data and handle the data request:

1. Write a code to request the reuired data in TEAMS's AOP language. The general data starts with "general" keyword, and dynamic data starts with the point-cut designator keyword. For example the following AOP code illustrated an example:

   ```
   aspect MyAspect {
      pointcut Call(): within(org.h2) && call(* get())
      {
      requires General.MethodName;
      requires Call.callee;
      };
      Advice1 before() : Call ;
   }
   ```

2. Generate the general data and essential arguments to generate dynamic data. The makeArgumentValue method should be implemented in your shadow matcher class for this reason. The following code shows how the method should be implemented for the loop example.

   ```
   public class LoopShadowMatcher implements PCDShadowMatcher {
       @Override
   ```

```
        public void makeArgumentValue(final JoinPointShadow joinPointShadow,
            final MethodNode methodNode,
            final Vector<JoinPointInfoExp> infoExpVect) {
            final StaticJoinPointInfo staticJoinPointInfo = new StaticJoinPointInfo();
            for (int i = 0; i < infoExpVect.size(); i++) {
               if (infoExpVect.get(i).getType().equalsIgnoreCase("general")) {
                  switch (infoExpVect.get(i).getName().toLowerCase()) {
                      case "methodname":
                         staticJoinPointInfo.setMethodNode(methodNode);
                      case "classname":
                        staticJoinPointInfo.setClassName(methodNode.getClass().getName());
               }
            }
         }
           joinPointShadow.setStaticJoinPointInfo(staticJoinPointInfo);
           final DynamicJoinPointInfo dynamicJoinPointInfo = new DynamicJoinPointInfo();
           for (int i = 0; i < infoExpVect.size(); i++) {
              if (infoExpVect.get(i).getType().equalsIgnoreCase("loop")) {
                 switch (infoExpVect.get(i).getName().toLowerCase()) {
                     case "time":
                        dynamicJoinPointInfo.setTimeArg();
                 }
              }
           }
           joinPointShadow.setDynamicJoinPointInfo(dynamicJoinPointInfo);
    }
    }
```

3.

5.4 Advice Execution Model

TEAMS supports before and after as advice execution model. The approach of extend and develop other AEMs will be provide in the next version.

The next version of TEAMS will be released on Jan 15, 2014.