IOWA STATE UNIVERSITY
Digital Repository

2008

On the test-driven development of emerging modularization mechanisms

Rakesh Bangalore Shivarudra Setty
Iowa State University

Follow this and additional works at: https://lib.dr.iastate.edu/etd

Part of the Computer Sciences Commons

On the test-driven development of emerging modularization mechanisms

by

Rakesh B. Setty

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Science

Program of Study Committee:
Hridesh Rajan, Major Professor
Andrew Miner
Tien Nguyen

Iowa State University

Ames, Iowa

2008

DEDICATION

To my parents without whose unflinching support, love and encouragement, I would not have been able to come this far.

TABLE OF CONTENTS

v

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to take this opportunity to express my sincere thanks to those who helped me with various aspects of conducting research and the writing of this thesis. First and foremost, Dr. Hridesh Rajan for his outstanding guidance, patience and support throughout this research and the writing of this thesis. His constant motivation and creative insights have often encouraged me and kept my research moving. His weekly research paper reading sessions presented a lot of opportunities for exploring newer domains in research and helped understand domains like programming languages and type theory better. I also thank him for sponsoring my travel to the AOSD'07 conference which helped me broaden my perspectives and present my work to the research community.

I would like to thank my committee members Dr. Andrew Miner and Dr. Tien Nguyen for their encouragement and help. I am very thankful to Robert Dyer for his collaborations in my research. He has lent his insights and technical skills during various stages of my research. He also proof-read a part of my thesis. I would like to thank all other members of the Laboratory of Software Design for constructive criticism and timely suggestions during research. They also gave a great friendly atmosphere to work.

I am very grateful to Dr. Kasthurirangan Gopalakrishnan , Dr. Siddhartha Khaitan, Ankit Agrawal and Mahantesh Hosamani for their unwavering support and encouragement which enabled me to make breakthroughs when required. I am very grateful to my parents Mr. K. C. Shivarudra Setty and Mrs. Poornima Devi, for their unwavering love and support. I thank my brother Vishwas Setty for his encouragement and moral support throughout my studies.

ABSTRACT

Emerging modularization techniques such as aspects and their precursors such as events in implicit invocation languages aim to provide a software engineer with better facilities to separate conceptual concerns in software systems. To facilitate adoption of these techniques in real world software projects, seamless integration into well-accepted practices such as a test-driven development process is essential. To that end, the main contribution of this thesis is an analysis (both pragmatic and theoretical) of the impact of a class of such techniques on the efficiency of a test-driven development process, which involves frequently compiling and testing programs in a process commonly known as the edit-compile-test cycle. I study two variants: the popular model of aspects as in the AspectJ-like languages, and a recently suggested alternative based on quantified, typed events embodied in the Ptolemy language. I present a case study analyzing two variants of the aspect-based model on two open source projects and a theoretical analysis of the quantified, typed event-based model. My results show that a seamless adoption of the aspect-based model requires careful balancing of competing parameters to ensure efficiency of a test-driven development process, whereas a quantified, typed event-based model naturally supports separate compilation thus decreasing the time spent in the edit-compile-test cycle.

CHAPTER 1. INTRODUCTION

Software systems are getting increasingly complex. According to Dijkstra, "to keep intellectual control over growing complexity one must be able to analyze a concern in software systems in isolation, for the sake of its own consistency" [17]. The need for improved separation of concerns is often addressed by invention of new modularization mechanisms [38]. Thus the development and refinement of modularization mechanisms (module systems) are of vital importance. A number of modularization techniques have been proposed in recent literature including but not limited to Units [20], Mixins [12], Open Classes [16], Roles [31], Traits [45], Implicit Invocation [23], Hyperslices [37], and Aspects [28]. Support of new modules is often provided as a language design and associated infrastructure.

A pragmatic issue for the empirical assessment of the benefits of the new modularization mechanisms is their adoptability in large projects, where improved separation of concern techniques are really needed to manage complexity. Large scale use puts the modularization mechanism to test and serves to illustrate its benefits (if any). However, such adoption depends significantly on the seamless integration of new modularization mechanisms and associated language infrastructure into well-accepted practices in software development. Test-driven development [8] is one such widely accepted practice.

Test-driven development techniques [8] have proven to be very effective in reducing defects and improving the overall quality of software systems [34]. The key characteristics of a test-driven development process (apart from the acclaimed difference that tests play a key role in the design of the system) is that tests are executed frequently. It is generally recommended that development proceed in small increments followed by rigorous testing of these increments (commonly known as the edit-compile-test cycle). The speed of this development technique directly affects developer's productivity in nontrivial systems [32] and a slow edit-compile-test cycle is reported to cost as much as forty percent of developer productivity [2].

Although many of these new modularization have been shown to have addressed the problem of separation of concerns in their own way, not much study has been done to see how well they support this widely accepted practice of test-driven development. In this thesis work, I make two specific contributions. I study the support provided by two most popular techniques that address the problem of separate compilation, namely aspect-orientation and implicit invocation.

In aspect-orientation, there are three popular deployment models, namely static weaving, load-time weaving and run-time weaving. In this work, I compare the support provided by static weaving and load-time weaving. More specifically, I analyze some of the parameters that differentiate between these two techniques and the roles that they play in differentiating. I choose AspectJ for its popularity and its maturity of implementation. But the results should apply to many other AspectJ-like languages.

For implicit invocation, I choose to study the support provided by a recently introduced language called Ptolemy [41]. It adapts some ideas from both aspect-orientation and existing implicit invocation languages. The key features of the language are its support for quantified, typed events and its support for declaring any arbitrary expression as an event. It has advantages over both aspect-orientation and implicit invocation languages. However, since it does not have a mature infrastructure at this point, I do a theoretical analysis of the language. More specifically, I study the property of separate compilation provided by the language and prove that it does have this property which is very important for programmer productivity as it allows multiple teams to work parallelly just by knowing each others interface. But from the point of view of test-driven development, separate compilation is a prerequisite for a good incremental compilation which potentially reduces the edit-compile-test cycle.

The rest of this thesis is organized as follows. Chapter 2 studies the support provided by AspectJ-like aspect-orientation (AO) languages. Specifically, it studies two popular deployment models in AO languages namely, static weaving and load-time weaving and determines some of the parameters that differentiate between these two techniques in supporting test-driven development and studies their roles in differentiating them. Chapter 3 studies the support that implicit invocation provides for test-driven development. It studies a recently proposed language called Ptolemy that has advantages over both aspect-orientation and implicit invocation. Since the language does not yet have a mature infrastructure, the chapter includes a theoretical analysis of the language, more specifically its support for

separate compilation which is an important property from the point of view of test-driven development. Chapter 4 compares and contrasts our work with related ideas. Chapter 5 describes future work and Chapter 6 concludes the thesis.

CHAPTER 2.   SUPPORT FOR TDD IN ASPECT-ORIENTATION

In this chapter, I study the support for test-driven development (TDD) provided by aspect-orientation. I study two of the popular deployment models in aspect-orientation (AO) namely, static weaving and load-time weaving. The aim of this chapter is to find which of these two techniques is better in supporting test-driven development. As I found out based on my experiments, which of these two techniques perform better depends on a number of parameters which I discuss and show the roles that they play in differentiating these two parameters. I also depict a possible research path which could help determine which of these two techniques perform better by knowing some of the values of these parameters. I choose AspectJ for its popularity as well as the maturity of its infrastructure. I begin by providing an introduction to the AO concepts and the deployment models in AO languages.

The objective of aspect-oriented programming (AOP) techniques is to aid software engineers by providing them improved mechanisms for separation of concerns [19, 29]. The most common way of modularizing these concerns are usually through a class-like module called *aspect*. Some of the common examples of concerns are security, quality of service, caching, buffering, etc.

2.1   Background on deployment techniques

I first introduce the concept of an aspect. To do that, I first explain the concepts that an aspect depends upon. A *join point* is a point in the execution of a program which is usually implicitly exposed by the semantics of the language. An example would be a call to a method. A *join point shadow* is the corresponding lexical point in the program. A *pointcut* is a predicate selecting a subset of join points in a program. An *advice* is a behaviour that is to be exhibited *before*, *after* or *around* the join points specified by a pointcut. Finally, an *aspect* usually declares some pointcuts and corresponding advices to them.

A dimension of interest for this work is what is called the *deployment model*. To understand the notion of deployment models, let us consider a requirement that may benefit from AO languages. Assume that we are adding a resource-sharing policy to an application, e.g. database connection sharing or thread pooling. To implement this policy, one can identify resource creation throughout the application and replace all such creation by a request from the resource pool, however, such an implementation strategy would couple all such parts of the application with the implementation of the resource-pooling requirement. The resource pooling implementation would be fragmented and spread across the program, which would make it difficult to evolve this implementation.

An AO solution to this problem would localize the implementation of the resource pooling policy into a module, use declarative constructs to identify resource creation throughout the application, and use another set of declarative constructs to override such creation with a resource pool request. This is done without leaving the boundary of the module, thereby solving the problem of evolution of the implementation. At some point, however, these declarative AO constructs will have to be composed (or weaved) with the application's original code such that at execution-time, the desired (interleaved) behavior is manifested. This is often called *deployment* of AO constructs.

AO languages support a variety of deployment models: *static* deployment, where the weaving happens during compilation, *load-time* deployment [30], where the weaving happens when a class is loaded for execution by a virtual machine (VM), and *runtime* deployment [39], where the weaving happens during an application's execution. Throughout this thesis, I use the terms deployment and weaving interchangeably. AO approaches support all three deployment models and each has its own advantages and disadvantages. For example, a static deployment model is likely to incur the cost of weaving once during compilation and thereafter the compiled application will run without the need for weaving. On the other hand, such a deployment model is unlikely to provide much dynamic flexibility, unless all such flexibility is anticipated and compiled into the application, which may incur additional overhead.

A load-time deployment model incurs the cost of weaving every time a class is loaded, however, it offers much more flexibility. The composition of AO code for resource pooling with the original code can be deferred until load-time. This, for example, allows one to start an application with or without

```
public class Hello {
 public static void main(String[] args){
   System.out.println("Hello");
 }
}
```

```
public aspect World {
  pointcut main(): execution(* Hello.main(..));
  after returning(): main() {
    System.out.println("World");
  }
}
```

Figure 2.1   A simple aspect-oriented application

resource pooling without having to recompile it. A runtime deployment model like AspectWerkz [11], Envelope based weaving [10], JAsCo [49], Nu [18], PROSE [39], or Steamloom [25] is likely to incur the cost of weaving during execution, although such costs can be significantly reduced [9, 18]. However, runtime weaving approaches allow one to defer the composition of AO code for resource pooling with the application until runtime. For example, one could introduce a resource pooling policy in a running web server if the resource availability is critically low and remove it subsequently when the underlying problems are solved, without having to restart the application [54].

As an example, consider a typical static weaving technique that is demonstrated using the simple AspectJ [28] application shown in Figure 2.1. I emphasize AspectJ for the maturity of its design and the availability of a robust and usable implementation, however, note that other aspect-oriented languages such as Eos [42, 43] would exhibit similar behavior. Also, ajc the compiler that we use is open source, very mature and most used compiler for AspectJ. The first release of this compiler was in 2003 and so far it has gone through 6 major releases. It has got very good attention from many developers to implement all possible features and optimizations. abc [7] is another popular compiler for AspectJ. But it does not support load-time weaving and also is generally slower than ajc since it is designed for research purposes than for regular use.

Our application has one class `Hello` (shown with a white background) and one aspect `World` (shown with a grey background). The class `Hello` declares a method `main` that prints the string `"Hello"` on the screen and exits. The aspect `World` declares that after the execution of the method `main` the string `"World"` will be printed. Here, the execution event "execution of the method main" is an example of a join point and the corresponding point in the code which corresponds to it is a join point shadow. The code `"execution(* Hello.main(..))"` which is the mechanism used to select the join point is a pointcut. The method-like construct that prints `"World"` is an advice.

2.1.1 Object Code View

```
public class Hello {
  static void main(java.lang.String[]);
0:  getstatic       #21;//Field System.out
3:  ldc             #22;//String Hello
5:  invokevirtual   #28;//Method println
    //Code inserted for aspect invocation
8:  goto            20
11: astore_1
12: invokestatic
#38;//Method World.aspectOf
15: invokevirtual   #41;//Method World.ajc$0
18: aload_1
19: athrow
20: invokestatic
#38;//Method World.aspectOf
23: invokevirtual   #41;//Method World.ajc$0
26: return
}
```

```
public class World {
  public static final World ajc$perSingletonInst;
  static {}; // Static initializer
0:  invokestatic #14; //Method ajc$postClinit
3:  goto            11
6:  astore_0
7:  aload_0
8:  putstatic     #16;//Field ajc$initFailureCause
11: return
//Advice ajc$0, constructor World, and methods
//hasAspect, aspectOf and ajc$postClinit elided.
}
```

Figure 2.2   An AspectJ aspect compiled to standard bytecode using static weaving: the generated code for the `World` concern is in gray

```
public class Hello {
  static void main(java.lang.String[])
0:  getstatic
#21;//Field System.out
3:  ldc
#22;//String Hello
5:  invokevirtual
#28;//Method println
8:  return
}
```

```
@PointcutDeclaration("execution(* Hello.main(..))")
public class World {
  public static final World ajc$perSingletonInst;
  static {}; // Static initializer
0:  invokestatic #14; //Method ajc$postClinit
3:  goto            11
6:  astore_0
7:  aload_0
8:  putstatic     #16; //Field ajc$initFailureCause
11: return
//Advice ajc$0, constructor World, and methods
//hasAspect, aspectOf and ajc$postClinit elided.
}
```

Figure 2.3   The example from Figure 2.1 compiled for load-time deployment: the generated code for the `World` concern is in gray

In our previous work [46], we compiled this *HelloWorld* application using the AspectJ compiler *ajc* using both static and load-time weaving. We disassembled the class files using *javap*, the disassembler for Java. Figure 2.2 shows the disassembled intermediate code for static weaving represented with Java bytecode notations.

The figure shows the disassembled code of `Hello` with a white background and the disassembled code of `World` with a grey background. As can be observed for static weaving, the intermediate code to invoke `World` at the join point shadows is inserted into the class `Hello` in the method `main`. As a result, certain changes in `World` will require re-compilation of the `Hello` class.

Now consider the same application, but compiled with ajc for load-time deployment. The standard bytecode representation, shown in Figure 2.3, is similar to Figure 2.2 with a few differences. The standard bytecode representation for load-time weaving is similar with a few differences. First, the `Hello` concern is free of code from the `World` concern. Second, the `World` concern has additional Java annotations attached to the class and methods that inform the load-time weaver of how to weave the `World` concern into other classes as they are loaded. The load-time weaver reads a configuration file (by default, named aop.xml in AspectJ) that indicates what aspects should be woven into the system. As a result of this compilation technique, a change in `Hello` will only affect the intermediate code representation of the `Hello` module. Similarly, a change in `World`, which is a crosscutting concern, will only affect the intermediate code representation of the `World` module. The changes are thus traceable to a limited number of modules at the intermediate code level, resulting in improved incremental compilation time compared to static deployment models. In fact, in our previous work [46, Section 3], we show that many of the changes in aspects like adding or removing an aspect, changing pointcut or an advice, adding, removing or modifying an inter-type declaration, etc trigger full build.

2.2 Case Study

As mentioned before, incremental compilation in static weaving in AspectJ-like languages seems to be a significant problem in test-driven development. Although, static weaving takes more time in compiling than load-time weaving, in load-time weaving approach, the execution time is longer than static weaving since the deferred composition has to take place while loading classes. Since the speed of a test-driven development process is dependent on the sum of compilation time and execution time, in this case study I analyze the trade-off in detail.

Based on my preliminary analysis, I set out to study four parameters that I determined to be relevant. These parameters are:

- the size of the project,

- the number of classes loaded during execution,

- the average join point shadow usage, and

- percentage of join point shadows advised.

Sections 2.2.5 - 2.2.9 explain these parameters and describe the results and analysis of the experiments conducted as part of this case study to examine the role of these parameters. Throughout this case study, the set of projects described in Section 2.2.1 were used. All experiments described in these sections were performed on Red Hat Enterprise Linux 4 with a 3.8 GHz Pentium IV processor and 3.5 GB of main memory.

The experimental methodology for Sections 2.2.5 - 2.2.8 is described in Section 2.2.2. The experiment described in Section 2.2.9 uses a slightly different methodology, which is presented there.

2.2.1 Candidate Projects

For the experiments conducted in Sections 2.2.5 - 2.2.9, I used two large open source projects: Apache Ant [6] and JBossCache [48]. The primary criteria for project selection were the presence of a reasonable and well-developed unit test suite and medium to large project size.

Apache Ant is a Java-based build tool. Ant has close to 800 Java source files with 93k lines of code and over 1000 compiled class files. It has a very well developed unit test suite, with close to 300 source files.

The JBossCache project aims to provide enterprise-grade clustering solutions to Java-based frameworks, application servers and custom-designed Java SE applications. JBossCache has almost 400 Java source files with 41k lines of code and 500 compiled class files. It also has a very well developed unit test suite, with over 400 source files. Although the project is smaller than Ant, it has a very good test infrastructure to conduct the experiments. The smaller size of the project (compared to Ant) also helps us see how the size of the project plays a role in distinguishing static and load-time weaving techniques.

My candidate projects have unit tests for only the object-oriented parts. However, this is not a threat to the validity of my experiments because I am only using very simple advice that can be verified by inspection. In the future, I could use the Raspect-like approach [55, 56] for automatically generating unit tests for newly introduced aspects in the candidate projects.

2.2.2 Experimental Setup

I run individual unit tests present in the Apache Ant [6] and JBossCache [48] projects which include an aspect that advises every join point shadow. I then measure the compile time, test time, and compile+test time for both static and load-time weaving. Before I run the next unit test, I rebuild the project. For static deployment, I use ajc $1.5.4$ and for load-time deployment, I use javac $1.5.07$ for the object-oriented source code and ajc $1.5.4$ with `XterminateAfterCompilation` option which disables weaving for the aspect-oriented source code. The performance with ejc compiler is similar that of javac compiler. Although I do not use incremental compilation for rebuilding, I do not use full build as well. I observe that if I do not clean the project before compiling it, it takes much less time than a full build does. Moreover, as we showed in our previous work [46, Section 3], most changes to aspects trigger a full build and hence our approach is not necessarily a threat to validity. One more thing to observe is that this time in rebuilding is measured only in Section 2.2.5.

Note that using javac for object-oriented source code (base code) in load-time weaving is not a threat to validity as I am comparing the performance of the techniques and not the performance of a specific compiler for the two techniques. In other words, I am comparing the best approach for static weaving to the best approach for load-time weaving. This should be seen as an advantage that load-time weaving inherently possesses against static weaving during compilation. I also verified that there is no significant statistical difference between the runtime performance of the bytecodes generated by javac and ajc respectively by running the test suite of Ant compiled by ajc and javac without any aspects. This suggests that there is no advantage or disadvanatage for one technique over the other in terms of runtime due to differences in the generated bytecodes.

I make use of the universal aspect as shown in Figure 2.4 to maintain the percentage of join point shadows covered by all unit tests constant at 100%. For example, assume that I have an aspect that covers 50% of the base code i.e. 50% of the join point shadows present in the base code are advised by the aspect. This does not mean that exactly 50% of the code in each unit test is covered by the aspect. The universal aspect that I use eliminates such a possibility.

For each unit test, I measure C - the number of classes that were loaded, J - the total number of join points executed, j - the average join point shadow usage (the ratio of the total number of join

```
1  public aspect UniversalAspect {
2  pointcut traceAll():(
3   call(* *.*(..)) || execution(* *.*(..)) || handler(*) || get(* *) || set(* *)
4   || initialization(*.new(..))|| preinitialization(*.new(..))
5   || staticinitialization(*)
6  ) && !(
7   within(*..UniversalAspect+)|| get(* *UniversalAspect.*)|| set(* *UniversalAspect.*)
8   || initialization(*UniversalAspect.new(..))
9   || preinitialization(*UniversalAspect.new(..))
10  || staticinitialization(*UniversalAspect)
11 );
12  . . .
13 }
```

Figure 2.4   The aspect UniversalAspect that traces all join point shadows

points executed to the total number of join point shadows loaded), and $NETR$ - the normalized ratio (explained in Section 2.2.4) of the execution times taken by the two techniques. I use ratio here and not difference since ratio gives us the relative performance of the two techniques which makes the results independent of the hardware on which the experiments were conducted.

I compile the project with the UniversalAspect using ajc with the showWeaveInfo option. When ajc is supplied with this option, it also outputs information about how weaving was done on the compiled files. In particular, information about join point shadows are emitted by the compiler. Using this information, I compute the join point shadows woven by the UniversalAspect.

To find out the values of the parameters, I used the advice shown in Figure 2.5. This advice records the class and the join point shadow that invoked the advice, making use of reflection API provided by AspectJ. It keeps track of all classes loaded, the join point shadows executed and the total number of join points executed. The advice writes the data to the file system at least once every 50 times it is invoked and hence there can be an error of at most 50 in computing the number of join points executed (J). This is an error margin of less than 1% in most unit tests for Ant which has smaller unit tests than JBossCache. On an average the margin of error introduced is much less than 1%. I did not choose to write the data on every execution of the join point due to the excessive time it took to run the experiments. Note that there is no such margin of error introduced while counting the number of classes loaded as whenever a new class is loaded, corresponding data is written by the advice.

I executed all unit tests with this advice to determine the number of classes loaded, the number of join points executed and the average join point shadow usage. To measure the test execution time, I

used a different advice that only does a dummy operation like incrementing a counter (several times) to avoid unpredictable factors like I/O. I used longer advice to ensure that the execution time of advice is much higher than the noise that could be introduced due to extraneous factors. As the size of the advice decreases, the role of noise in the results increases. However, the trends noticed in the results should remain the same irrespective of the length of the advice.

```
1 before(): traceAll() {
2   // jpMap is a hashtable mapping class names to a hashset consisting
3   // of all join point shadows within the class that were executed.
4   String jpName = thisJoinPoint.getSourceLocation().getWithinType().getName();
5   if(jpMap.containsKey(jpName)) {
6     HashSet jpSet = ((HashSet)jpMap.get(jpName));
7     if(jpSet.add(thisJoinPoint.toString())) count++;
8   } else {
9     HashSet jpSet = new HashSet();
10    jpSet.add(thisJoinPoint.toString());
11    count++;
12    jpMap.put(jpName, jpSet);
13  }
14  //jps_executed keeps track of the number of join points executed
15  jps_executed++;
16  //classCount keeps track of the number of classes loaded during execution
17  if((count >= 50) || (jpMap.size() >= classCount+1)) {
18    if((jpMap.size() >= classCount+1)) {
19      count = 0;
20      classCount = jpMap.size();
21      dumpTable(); //saves info in jpMap as well as jps_executed
22    }
23  }
24 }
```

Figure 2.5   The   advice   for   the   `traceAll`   pointcut   in   the   aspect `UniversalAspect`

2.2.3   Notations Used in the Analysis

The following are the notations used in the experiments that I conducted to find the impact of the parameters involved.

C – the total number of classes loaded during execution

c – the ratio of the number of classes loaded to the number of classes present

J – the total number of join points executed

j – the average join point shadow usage

NETR – the normalized execution (of test) time ratio defined as the ratio of the normalized test times taken by load-time and static weaving techniques respectively (see Section 2.2.4)

2.2.4 Normalization Factor

In comparing the times taken by load-time and static deployment, there are some factors that have similar impact on both of them. This distorts the impact observed when varying the parameters. One of them is the *join point shadow density*. I define join point shadow density as the ratio of the number of join point shadows per line of executed code. To see the importance of this factor, consider the following example. Assume two methods that do not read/write to the fields of the class and do not call other methods (note that local variable reads and writes are not join points): one that does significant amount of computation using only local variables and the other that computes less. Both these methods will have same number of join points, the call and execution of the method, but their test execution times will differ. However, if we consider the difference in execution times with and without the aspect for the technique that we are considering, which I call normalized test times, they will be similar. Another such factor is the execution time of different types of join point shadows (e.g., method or field). In general, I saw cases where the execution times were less when there were more join points executed and vice-versa. Hence, I use the execution time without the aspect as a normalization factor to rule out the effect of the extraneous factors mentioned here.

2.2.5 Impact of Size of Projects

The size of the project is measured in terms of the total number of join point shadows present in the system. This parameter affects the compilation time taken by the two techniques. In our previous work [46], we showed that Eclipse takes much longer to compile (with static weaving) than Azureus because Eclipse is larger. As I will show later in my experiments, static weaving takes considerably longer compilation time than load-time weaving. Remember that for load-time weaving, I used javac to compile object-oriented source code and ajc to compile aspect code. This technique leads to much better incremental compilation while avoiding weaving during compilation. On the other hand, static

weaving should make use of ajc which has inferior incremental compilation due to the burden of weaving during compilation.

This parameter is the only one that helps load-time weaving perform better than static weaving. In other words, the time saved in compilation due to the size of the project is the only time that load-time weaving can gain over static weaving. During execution, load-time weaving cannot take less time compared to static weaving since load-time weaving will involve the overhead of weaving classes as they are loaded.

2.2.5.1 Results

Figures 2.6 and 2.7 show how each individual unit test performs with respect to the compile+test time for Ant and JBossCache, respectively. The X-axis represents the various unit tests present in the test suite. The Y-axis represents the compile+test time taken by each unit test. Note that the scale for the Y-axis is logarithmic since we are only interested in knowing which technique is better for a given unit test and not the exact time taken by each technique. The logarithmic scale also makes comprehending the results easier as we will be less distracted by the difference between larger and smaller unit tests. Also note that the bars representing the total time taken by a test run based on load-time deployment is super-imposed on the corresponding bars representing the total time taken by the same unit test run based on static deployment. If the bar corresponding to static deployment is not visible for any unit test, it means that static deployment performed better for that unit test.

Figure 2.8 shows how the two techniques, load-time weaving and static weaving, fare for Ant and JBossCache projects. As mentioned earlier, Ant is almost double the size of JBossCache. The Y-axis represents the total time taken by unit tests.

2.2.5.2 Analysis

Observation 2.2.1 *For individual unit tests, with respect to compile+test times, load-time weaving performs better for most unit tests of Ant which is a bigger project with smaller unit tests, whereas static weaving performs better for most unit tests of JBossCache which is a smaller project with bigger unit tests.*
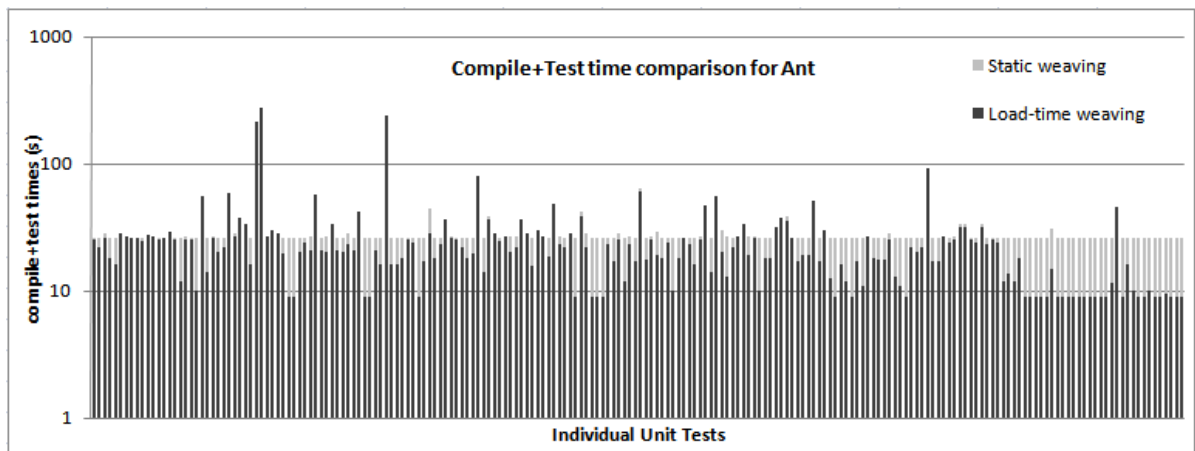
Figure 2.6   The comparison of the compile+test times for Ant. Observe that load-
-time weaving performs better for most unit tests

The main reason for longer compilation time in static deployment is the requirement to weave all the advices into the base code. Because of this, ajc does not have a very good incremental compilation ability. Since load-time weaving does not require weaving during compilation, it can make use of a compiler like javac which has very good incremental compilation ability to compile object-oriented code separately and compile only aspect code using ajc without weaving. From Figure 2.6 and Figure 2.8, we can see that load-time weaving performs better for most unit tests in Ant. Ant is a bigger project than JBossCache and so needs more time for compilation in static deployment and hence bigger size of project is an advantage for load-time weaving.   Also, the unit tests for Ant are smaller compared to JBossCache which means that the difference in execution time between load-time and static weaving in Ant is lesser compared to JBossCache in absolute terms. All the unit tests which performed better under load-time weaving had a larger difference in execution time (for both Ant and JBossCache) than the difference in compilation time. However, in the case of JBossCache, the size of the project is small and hence the advantage gained by load-time weaving during compilation is less. In addition, JBossCache has bigger unit tests which means that load-time weaving has a larger disadvantage during execution. These two factors combined resulted in most unit tests in JBossCache performing better under static weaving, as can be seen in Figure 2.7 and Figure 2.8.

Observation 2.2.2 *The time taken to weave a class while loading is significantly higher than that taken*
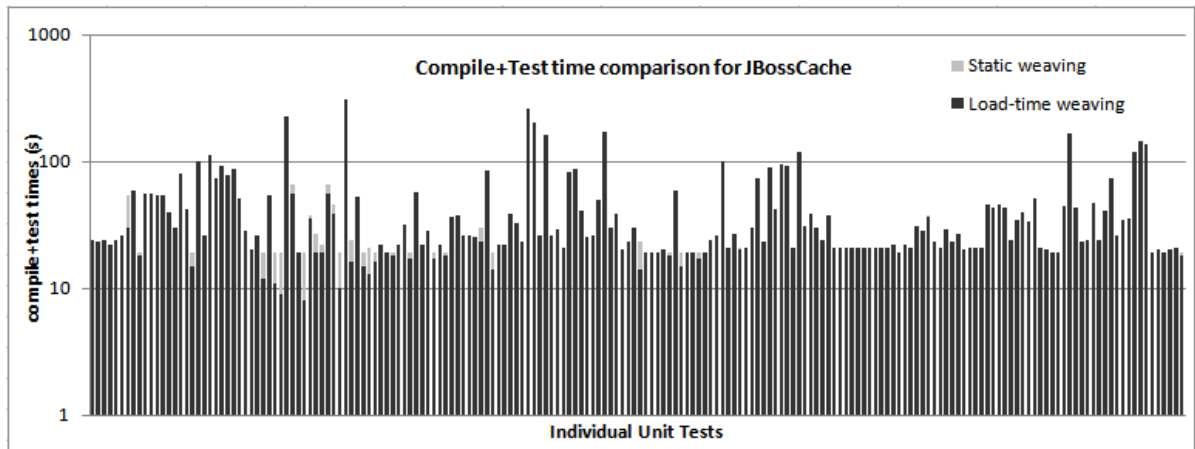
Figure 2.7 The comparison of the compile+test times for JBossCache. Observe
that static weaving performs slightly better for most unit tests

for the same class during compilation by static weaving.

Load-time weaving has to load only those classes that get executed. So, it usually weaves only a subset of existing classes. It also always saves some time during compilation. But it still sometimes take more time overall than static weaving. The only explanation for this is that the time to weave a class while loading is higher than during compilation by static weaving. It should be significantly higher because we observed that for bigger unit tests, the absolute difference between load-time and static weaving in compile+test time is considerably higher, sometimes load-time weaving takes more than double than that by static weaving. Also, we will see in next two sections that the normalized execution time ratio of load-time weaving to static weaving is usually in the range of hundreds. This is perhaps due to lack of some of the optimizations in load-time weaving where one class is woven at a time compared to those present in static weaving where all classes are collectively woven.

2.2.6 Impact of Number of Classes Loaded

The number of classes loaded (C) is an important parameter that affects the execution time taken by load-time weaving. The custom class loader (or Agent for Java 5.0 or greater) employing load-time weaving technique has to check each *loaded* class, while it is being loaded to determine, if the join points in the class are being advised by any aspects in the system. If a join point is determined to be
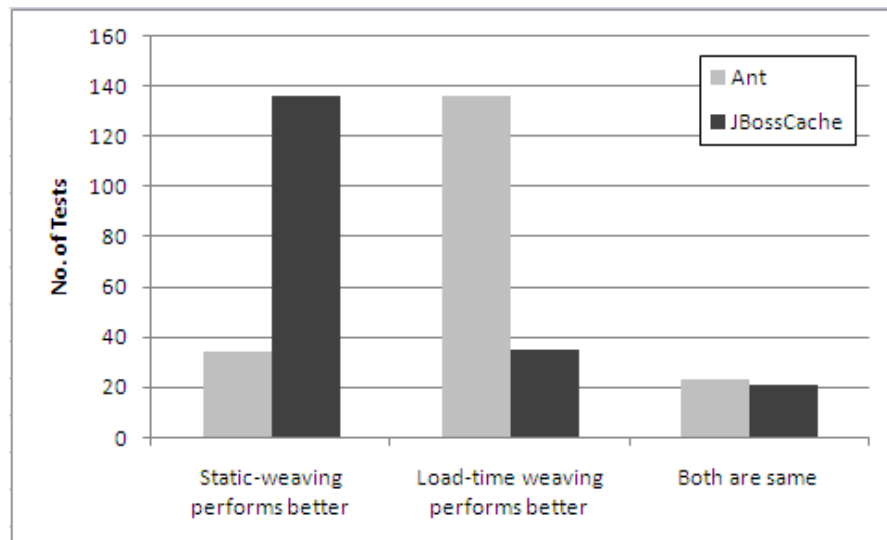
Figure 2.8   The comparison of the two techniques for Ant and JBossCache. The size of Ant is almost double the size of JBossCache and execution times of most unit tests in Ant are smaller compared to unit tests in JBossCache. Observe that load-time weaving performs better for most unit tests in Ant, while static weaving performs better for most unit tests in JBossCache.

advised by one or more aspects, the byte code representation of the join point is modified to invoke the advice in the aspects. This task is in addition to the tasks of a regular class loader. Recall that the static weaving technique performs similar matching for each class in the system during compilation. Thus intuitively it would appear that, the execution time taken by load-time weaving should clearly increase as the number of classes loaded increases. In terms of test-driven development, this intuition would translate to: edit-compile-test cycle of a module that utilizes or depends on a significant number of other modules is likely to be better off using static weaving. The results described in this section show that this intuition may not be entirely accurate.

2.2.6.1   Results

Figure 2.9 shows the normalized test times taken by the compilation techniques as the number of classes loaded (C) increases. Each data point in this chart represents a unit test of the two projects, namely Ant and JBossCache. The X-axis in the figure represents the number of classes loaded and the Y-axis represents the normalized execution time ratio ($NETR$). As described previously, $NETR$ is

the ratio of normalized test time taken by load-time weaving to the test time taken by static weaving.
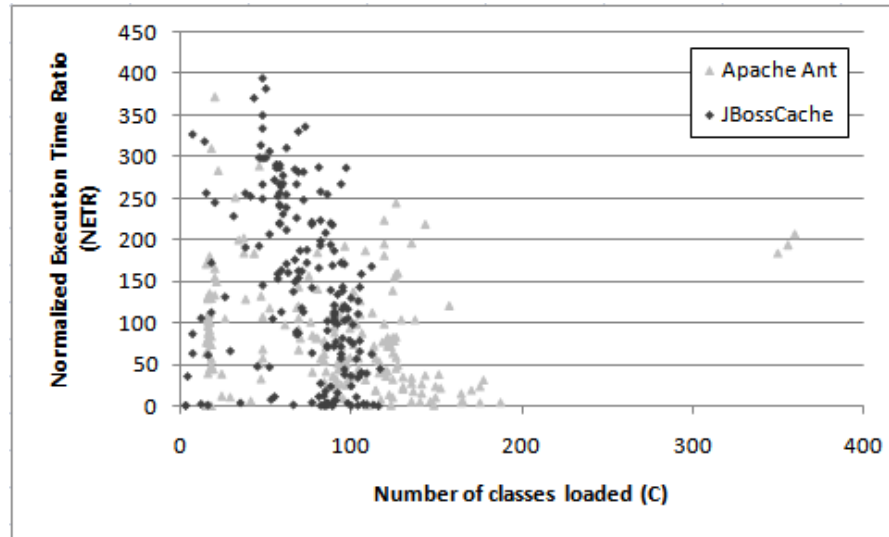
Figure 2.9   The effect of the number of classes loaded C on the normalized test
times taken by the two techniques for Ant and JBossCache

2.2.6.2   Analysis

In load-time weaving, the classes are woven as they are loaded. This is advantageous since typically only a subset of all the classes are loaded, meaning that only those classes are woven. This is also disadvantageous as this weaving is an overhead while running the unit tests. Overall, it is advantageous for load-time weaving when the number of classes loaded is less as that means fewer classes are woven. However, as the classes loaded increases, the overhead at runtime increases. Hence static weaving performs better as the number of classes loaded increases.

Figure 2.9 does not clearly support this due to the role of another parameter - the number of join points executed. Note that each unit test varies not only by the number of classes loaded, but also by the number of join points executed. Although I saw that in general that as the number of classes increased so did the number of join points executed, the number of join points executed varied based on the unit test.

We can compensate the role played by this parameter by considering the the ratio C/J instead, which represents the number of classes loaded per every join point executed.

Figure 2.10 The effect of the ratio C/J on the normalized test times taken by the two techniques for Ant and JBossCache. Observe that as C/J increases, the normalized execution time taken by load-time weaving gets worse compared to the static weaving

Figure 2.10 shows how the normalized test times taken by the compilation techniques vary as the ratio C/J varies. Like the previous figure, each data point in the graph represents a unit test of the project indicated by the corresponding legend. The X-axis in this graph represents the ratio C/J and the Y-axis represents the $NETR$.

The graph shows that as the value of C/J increases, the normalized execution time ratio increases. This means that load-time weaving takes longer compared to static weaving. The ratio C/J thus appears to be a better differentiating factor between the two techniques.

We can see that the value of $NETR$ is in the range of hundreds, hence further strengthening the Observation 2.2.2.

We can see some outliers in Figure 2.10. Most of these outliers are the unit tests which are very small. For example, the outlier in the far right of the graph which represents a JBossCache unit test with C/J value of over 0.12 is extremely small. It corresponds to the unit test C3p0ConnectionFactoryTest in the package org.jboss.cache.loader which loads just 7 classes and executes only 65 join points. In this particular case, the unit test compiled by ajc actually took 78 milli-seconds less to execute than that generated by javac. This is because noise introduced in such cases becomes comparable to the

normalized execution time. Other unit tests that are far away from the trend line either have just a few hundred of join points executed and/or less than 25 classes loaded in general.

Note that the impact of this parameter will remain the same irrespective of the size or type of project since it only affects the test time. Our conclusions are obtained based on many unit tests ranging from very small to very large.

Observation 2.2.3 *As the number of classes loaded increases, the test time taken by load-time weaving increases faster than static weaving.*

2.2.7 Impact of Average Join Point Shadow Usage

The average join point shadow usage (j) is also a parameter that affects the normalized execution time ratio ($NETR$). It tells us the number of times every join point shadow loaded gets executed on average. As this number increases, the overhead involved in weaving the classes during loading (which is proportional to the number of join point shadows loaded) becomes smaller compared to the total time taken to execute. However, the execution time of load-time weaving will always be higher than that of static weaving, i.e. the ratio of times taken will never be less that 1. The ratio of the execution times moves closer to 1 as the average join point shadow usage increases.

2.2.7.1 Results

Figure 2.11 shows how the normalized test times taken by the compilation techniques vary as the average join point shadow usage vary. Each data point in the graph represents a unit test of the project indicated by the corresponding legend. The X-axis represents the average join point shadow usage. The Y-axis again represents the normalized execution time ratio ($NETR$).

2.2.7.2 Analysis

Observation 2.2.4 *As the average join point shadow usage increases, the test time taken by load-time weaving approaches that taken by static weaving.*

The average join point shadow usage (j) also plays a role in determining which technique performs better. As this number increases, the overhead in loading a class becomes less compared to the time

Figure 2.11   The effect of the average join point shadow usage (j) on the normalized test times taken by the two techniques for Ant and JBossCache. Observe that as j increases, the normalized execution time taken by load-time weaving gets closer to that of static weaving

taken in executing the unit test, i.e. the time taken to run the unit test by load-time weaving becomes closer to that of static weaving. Figure 2.11 supports this hypothesis by showing that load-time weaving gets closer to static weaving as j increases.

Again, we can see that the value of $NETR$ is in the range of hundreds, hence further strengthening the Observation 2.2.2.

We can see some outliers in Figure 2.11. Most of these outliers are the unit tests which are very small. For example, the unit test C3p0ConnectionFactoryTest that we saw in the previous section represents the left-most outlier in the graph with value of j being close to 0.2 and value of $NETR$ being close to 64. Other unit tests that are far away from the trend line are also small in general.

Note that the impact of this parameter will remain the same irrespective of the size or type of project since it only affects the test time. Our conclusions are based on many unit tests ranging from very small to very large.

2.2.8   Impact of C and J on test-times

We have seen that both the number of classes loaded and the average join point shadow usage are parameters that differentiate between static weaving and load-time weaving techniques. But it would be interesting to find a single parameter (consisting of these two parameters) that will differentiate these two techniques during execution.

Normalized execution time for a given test is basically the extra time that the deployment technique takes over the normal execution of the test without any advice. This normalized execution time consists of two parts. The first part is the time taken to load the classes (with or without weaving). Note that the time taken to load classes in these deployment techniques will be larger due to the advice that gets woven into them. The second part is the time taken to execute the advices. I observe that the time taken to load classes is proportional to the number of join point shadows loaded and the time taken to execute the advices are proportional to the number of join points executed. The number of join point shadows loaded is the product of the number of classes loaded and the average size of a class (specific to the project). Therefore we can estimate the normalized execution time of a unit test as $m * J + n * C * s$, where m and n are constants and s is the average size of a class. The size of a class can be measured by the number of join point shadows in the class.

These constants m and n are dependent on other non-differentiating factors like the percentage of join points covered by the aspect code, join point shadow density, etc. Since the only difference between the two techniques are in the way the classes are loaded, we can expect m to be the same for both the techniques, but the value for n will be different for the two techniques. Since load-time technique involves weaving the classes just before loading, we can expect the value for n for load-time weaving to be much higher than for the static weaving technique. Now, we define the term approximate execution differentiator AED as follows:

$$AED = \frac{m * J + n * C * s}{m * J}$$

The AED is a parameter that helps differentiate between the two techniques while combining the roles of the parameters that it is composed of, i.e. both the number of classes loaded and the number of join points executed.
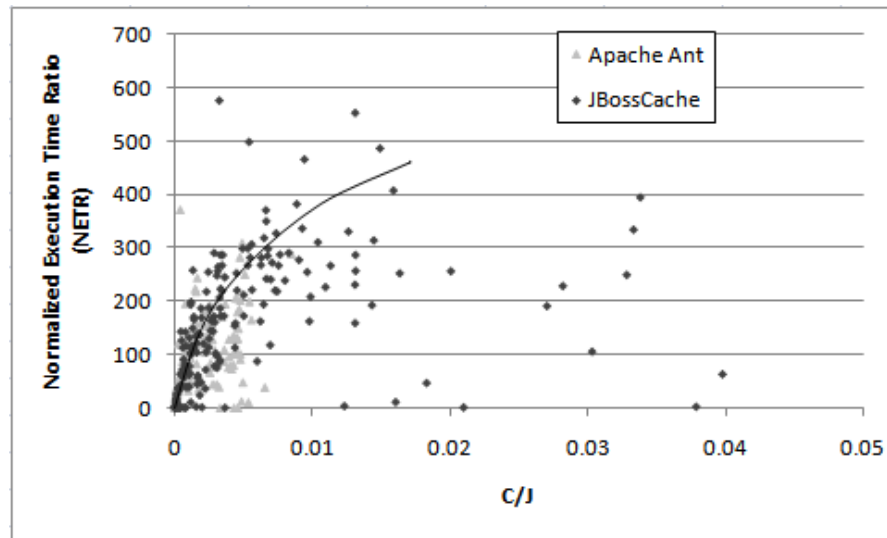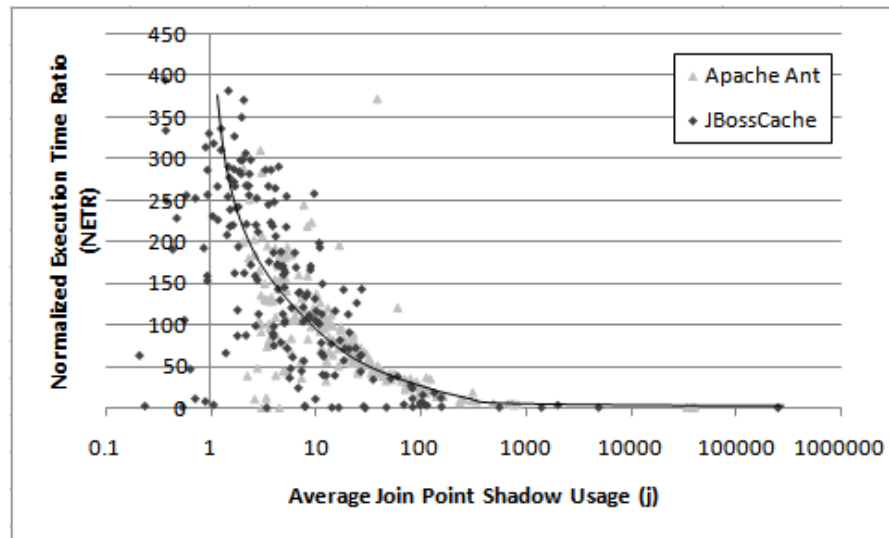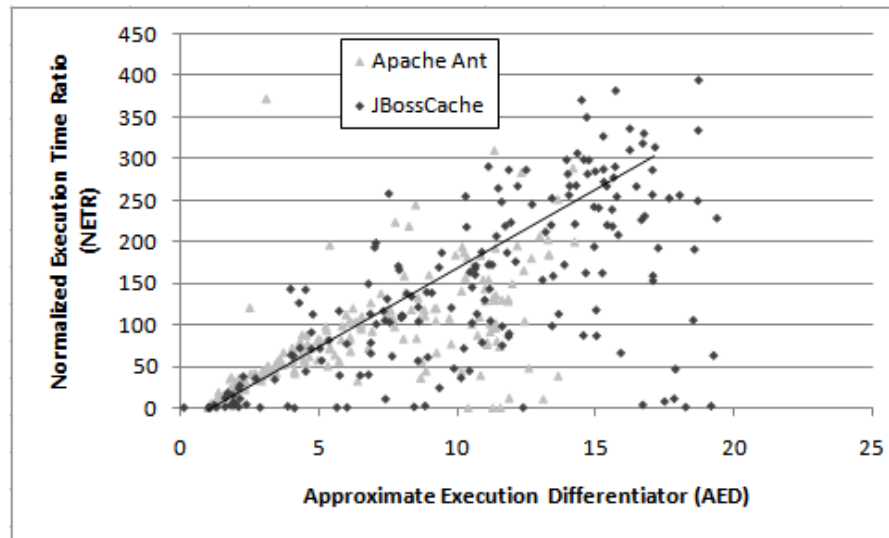
Figure 2.12 The effect of the AED on the normalized test times taken by the two techniques for Ant and JBossCache. Observe that as AED increases, the normalized execution time taken by load-time weaving gets worse compared to the static weaving

2.2.8.1   Results

Figure 2.12 shows how the normalized test times taken by the compilation techniques vary as the AED varies. For the purpose of this graph, the constants m and n are taken as 1 and 100 respectively, although the trend remains the same irrespective the values chosen for them. Each data point in the graph represents a unit test of the project indicated by the corresponding legend. The X-axis represents the AED. The Y-axis again represents the normalized execution time ratio ($NETR$).

2.2.8.2   Analysis

Observation 2.2.5 *As the AED increases, the test-time taken by load-time weaving increases faster than static weaving.*

The hypothesis that the AED is an indication of the ratio of the test-times of load-time weaving to static weaving is supported by Figure 2.12. Although this graph makes use of values 1 and 100 for the constants m and n respectively, we observed that varying the values for m and n does not change the trend, i.e. the trend remains a straight line but with a different slope.

We can see some outliers in Figure 2.12. Most of these outliers are the unit tests which are very small. For example, the unit test C3p0ConnectionFactoryTest that we saw in the previous sections represents the right-most outlier in the graph with value of the AED being close to 20 and value of $NETR$ being close to 64. Other unit tests that are far away from the trend line are also small in general.

Note that the impact of this parameter will remain the same irrespective of the size or type of project since it only affects the test-time. Our conclusions are obtained based on many unit tests ranging from very small to very large.

2.2.9 Impact of Percentage of Join Point Shadows Advised

It is natural to wonder if varying the aspect-oriented source code helps in differentiating the techniques. We can vary the kind of pointcuts and the percentage of object-oriented sources covered by varying the aspect. The percentage of object-oriented source code covered is the percentage of join point shadows in the object-oriented source code (base code) that is advised by the aspects present in the project. In this experiment, we will see if varying the aspect-oriented source code helps in differentiating between techniques.

2.2.9.1 Experimental Setup

In this experiment, we add an aspect to the source code of Ant and JBossCache. We keep modifying the pointcut such that it covers different percentages of join point shadows ranging from 2% to 100%. Using the `showWeaveInfo` option of ajc helped us to determine the number of join point shadows covered by the aspect during compilation. For each value of the percentage of join point shadows covered, we compile (full build, by which I mean cleaning followed by a build) the project and run the whole test suite (not each individual unit tests separately, unlike in previous experiments) that is part of the project and compare both the compile time as well as compile+test time of static and load-time weaving techniques. Observe that the regression test selection techniques proposed in [57, 58, 59] do not apply in this case since our aspects affect all the classes in the object-oriented source code even as it varies the percentage of join point shadows covered. Moreover, our objective of this experiment is to study the role played by the aspect code and not to find the best selection of unit tests to test a change

in code.

2.2.9.2   Impact of join point shadow coverage on the compilation times

Figures 2.13 and 2.14 compare the compilation times as the percentage of join point shadows covered by the aspect is varied for Ant and JBossCache. Here, the X-axis represents the percentage of the join point shadows affected by the advice. The Y-axis represents the time taken in seconds to do a full build of the project with the advice. The lower the time, the better the technique.

Figure 2.13   The comparison of compilation times for Ant as the percentage of join point shadows covered is varied. Observe that as the percentage of join point shadows affected increases, the compilation times taken by static weaving increases whereas for the load-time weaving it remains roughly the same

From Figures 2.13 and 2.14, we can see that load-time weaving performs far better than static weaving for all cases. Also, the compilation time for load-time weaving remains pretty much the same irrespective of the percentage of join point shadows covered by the aspect. This is because in load-time weaving, pure Java classes are compiled by javac and the aspect is compiled separately by ajc without any weaving. Hence changing the aspect does not affect the compilation of the Java classes. Also, the compilation time for static weaving increases linearly as the percentage of join point shadows covered by the aspect increases. This is because of the advice being weaved in to the classes during compilation itself. However, we can see few outliers which deviate from the trend very slightly.

Figure 2.14 The comparison of compilation times for JBossCache as the percentage of join point shadows covered is varied. Observe that as the percentage of join point shadows affected increases, the compilation times taken by static weaving increases whereas for load-time weaving it remains roughly the same

2.2.9.3   Impact of join point shadow coverage on the total times

Figures 2.15 and 2.16 compare the compile+test times as the percentage of join point shadows covered by the aspect is varied for Ant and JBossCache. Once again, the X-axis represents the percentage of the join point shadows affected by the advice. The Y-axis represents the time taken in seconds to do a full build of the project with the advice and run the entire test suite associated with the project. Again, the lower the time, the better the technique.

Observation 2.2.6 *For regression testing, both Ant and JBossCache confirm that percentage of join point shadows covered has no role in determining which technique performs better.*

From Figures 2.15 and 2.16, we can see that static weaving always performs better than load-time weaving. This is in spite of load-time weaving performing better during compilation. Note that in this experiment, a full build is done instead of just a recompile and the whole test suite is run at once without any compilation in between. Hence, the time saved by load-time weaving during compilation is negligible compared to the overhead involved while running the whole test suite. We can see that the compile+test time keeps increasing linearly for both static and load-time weaving techniques till the percentage of join point shadows is about 45% and then there is a fall followed by a similar increase.
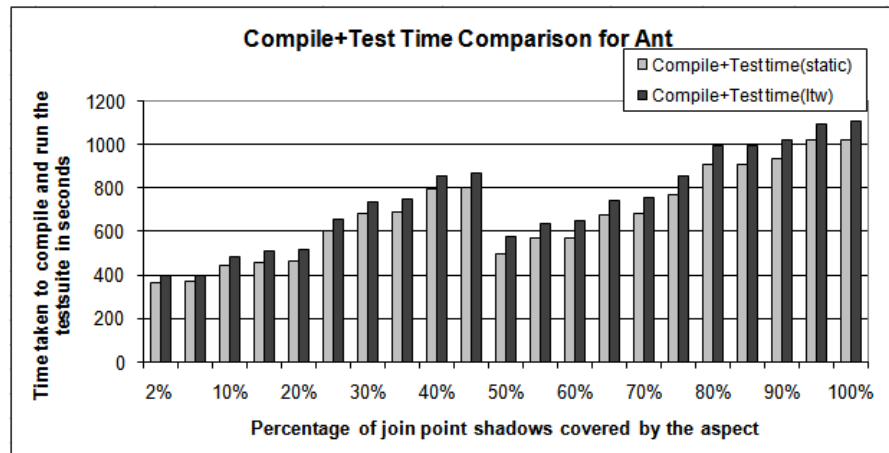
Figure 2.15 The comparison of compilation+test times for Ant as the percentage of join point shadows covered is varied. Observe that even as the percentage of join point shadows affected varies, the total time taken by load-time weaving always remains higher than that of static weaving.

When we modify the aspect to cover more join point shadows, not all the join point shadows covered by the previous aspect remain covered in the modified aspect, although the modified aspect will have more number of join points covered. This could mean that even though the percentage of join point shadows is increased, it could be the case that the percentage of join point shadows covered in the classes that are actually tested decreased. For example, in the case of the Ant project, the pointcut that covers close to 45% of join points includes all kinds of pointcuts except for method calls. But the pointcut which covers close to 50% of join points includes only method calls except for those which have a single argument of integer type. Now although these pointcuts cover 45% and 50% of the join points respectively, they cover very different set of join points. The test suite for Ant must be containing more join points that are covered by the first pointcut than those covered by the second pointcut and hence the execution time of the test suite with the first pointcut is more than that by the second pointcut. But what matters to us is that static weaving consistently performs better irrespective of the percentage of join point shadows covered by the aspect. Note that all the other parameters - size of the project, number of classes loaded and the number of join points executed are constant throughout. This shows that the aspect-oriented source code does not help in differentiating between the two techniques.
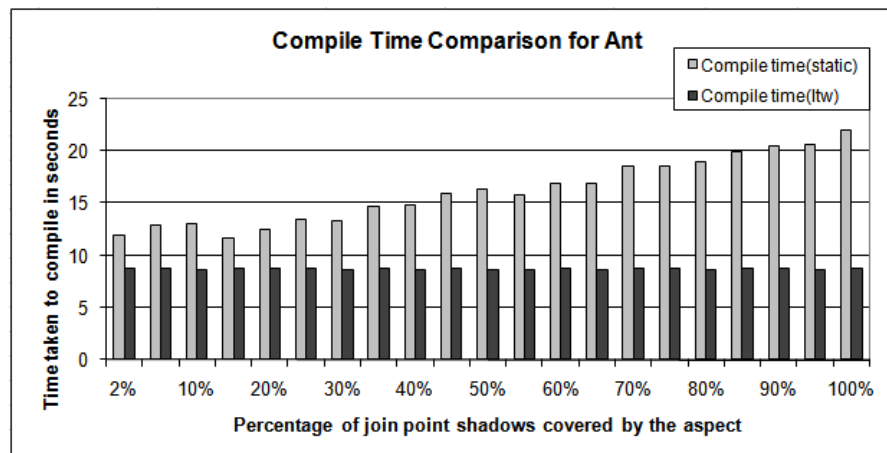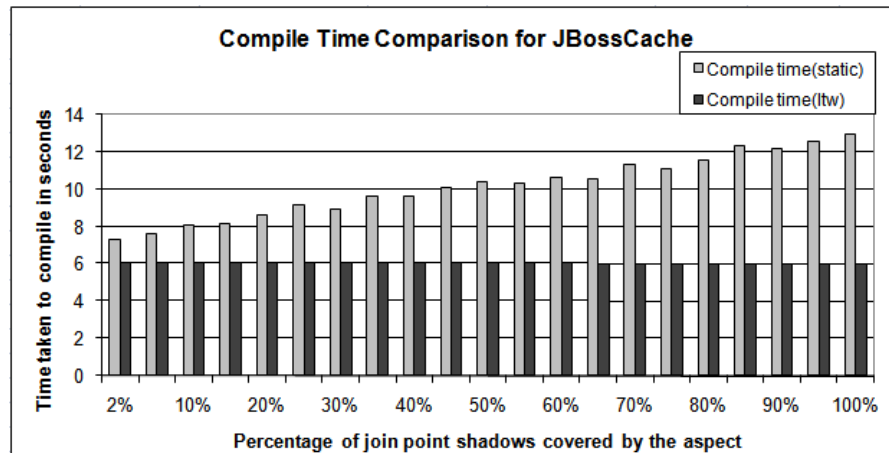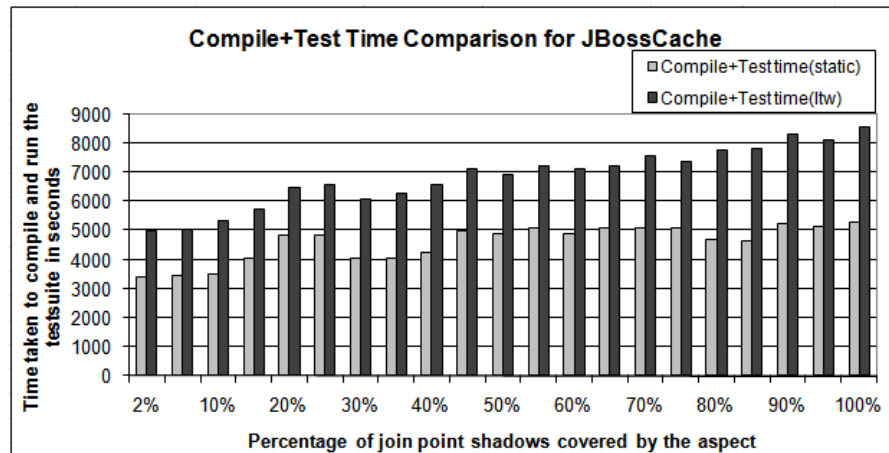
Figure 2.16   The comparison of compilation+test times for JBossCache as the per-
centage of join point shadows covered is varied. Observe that even as
the percentage of join points affected varies, the total time taken by
load-time weaving always remains higher than that of static weaving.

Load-time weaving might have performed better for some values of percentage of joinpoint shad-

ows covered for individual tests instead of the whole testsuite, but what we are interested in is whether

the technique which performs better changes by changing the percentage of joinpoint shadows covered.

As can be guessed from the Figures 2.15 and 2.16 (and confirmed by the experimental data), the ratio

by which one technique performs better over the other technique as the percentage of joinpoint shadows

covered is varied remains pretty much the same. This is because as the percentage of joinpoint shadow

covered is increased, the compilation time for static weaving increases which is often equally negated

by the overhead in weaving by load-time weaving.

2.3   Discussion

In my case study we saw that larger projects, lesser fraction of classes loaded and higher average

join point shadow usage are better for load-time weaving (cf. Sections 2.2.5 - 2.2.7). We also ob-

served that varying the join point coverage of the aspect code does not differentiate between the two

deployment techniques (cf. Section 2.2.9).

The results described in Sections 2.2.5 - 2.2.8 are based on a statistically significant population of

approximately 400 unit tests of varying size across the Ant and JBossCache projects. Each unit test

provides different values for the parameters C and J which help in differentiating the two techniques. My results are based not on the design of the tests but on the values that they provide for paramters that we study. I find that the test suites that experiment with provides widely varying values for the parameters that I study. For example, the value of C provided by unit tests in JBossCache varied from a minimum of 3 to a maximum of 116. Similarly the value of J provided by unit tests varied from a minimum of around 450 to a maximum of over 6 million. Similar range of values for observed for Ant as well. I believe that these results are representative of the typical behavior observed in an AO test-driven development process. The size of the project determines the difference in compilation time between the two techniques. Except for compilation time, the rest of the results described in this study, depend only on the characteristics of the unit tests run and not on the project itself. Thus I believe that the project selection itself does not play much role in the conclusions of this study. As a result, the trends observed in this case study can be easily extrapolated to other AO projects.

I found that running a unit test in Ant with our aspect can take about 8 seconds on an average and up to 32 seconds more if the better deployment technique is not chosen. Similarly for JBossCache, it could take about 5 seconds on an average and up to 75 seconds more if the better deployment technique is not chosen. Considering the development process is usually on-going with the edit-compile-test cycles repeating over and over again, this is a very significant saving worth considering.

2.3.1   Decision to Use a Deployment Technique

In the environment of a test-driven development process, neither static weaving nor load-time weaving outperforms the other in every single case. Some unit tests perform better under static weaving and others perform better under load-time weaving.

The factors which determine which deployment technique performs better are parameters like the size of the project, the number of classes loaded and the average join point shadow usage by the unit test. Static weaving has an overhead during compilation whereas load-time weaving has an overhead during execution. Which of these overheads is bigger is determined by these three parameters.

This case study only studies which deployment technique will benefit as the values of these three parameters are varied, but not which deployment technique should be used based on specific values of

these parameters. This direction of future work is discussed later in Chapter 5. Informally, for a large project with small unit tests, load-time weaving is likely to be better as I saw for most unit tests in Ant. On the other hand, for a smaller project with bigger unit tests, static weaving is likely to be better as I saw for most unit tests in JBossCache.

Which deployment technique to use during the test-driven development can be decided at three levels of granularity - project level, module level and unit test level. The decision at the project level is the coarsest level of granularity and the easiest, as it has to be made just once for all unit tests in the project. But the decision will not likely be optimal for every unit test, as each may differ with respect to two parameters - number of classes loaded and the average join point shadow usage. However for some projects, most unit tests could perform better under one of the deployment techniques like JBossCache in which most unit tests perform better under static weaving. In such cases, making the decision to use a deployment technique at this level of granularity will save a lot of time and effort.

The next level of granularity is the module level granularity, where the choice to use a deployment technique for test-driven development is made for each module. This is perhaps also a more practical trade-off between the time and effort spent on choosing a deployment technique and the time saved during development if an optimal technique is selected for each unit test.

The finest level of granularity is the unit test level granularity. Here an optimal deployment technique is chosen for each unit test based on the values of each parameter. Although this level of granularity ensures that each unit test is run with the optimal deployment technique specific to the unit test, it could involve lot of time and effort to determine such a deployment technique for each unit test. We have not investigated these techniques in this work and it seems that it may be impractical in general to support the fine granularity in choosing deployment technique at the level of individual unit tests.

Finally, our case study shows that static deployment model is always better compared to load-time deployment model in terms of just execution time. These results statistically shows that unless some design-level advantages exist for load-time weaving, static deployment should be used for the final build to produce production systems as suggested by many experts.

2.3.2 Broader Perspective

We showed in our previous work [46, Section 3] that incremental compilation is a problem in static weaving. Here, we see that load-time weaving does not completely solve the problem. The answer to the question of which deployment technique to use during test-driven development process is not straight-forward and depends on the parameters that we discussed. But incremental compilation and slower test-driven development process is just one of the various disadvantages of most aspect-oriented languages. AspectJ-like languages have other disadvantages like fragile pointcuts, quantification failure, limited access to context information, non-uniform access to irregular context information, lack of separate compilation among others. An existing alternative today is implicit invocation(II) languages. However, they have a different set of disadvantages like observers being tied to the code of subjects, lack of quantification and lack of around advices among others. Recently, Ptolemy [41] was introduced that claims to combine the best of both AO and II languages. In the rest of this thesis, I study the features of the language and prove that it supports separate compilation as well which is an important property from the perspective of programmer productivity in test-driven development methodology.

CHAPTER 3.   SUPPORT FOR TDD IN IMPLICIT INVOCATION

In this chapter, I study the support for test-driven development (TDD) provided by implicit invocation. I choose to study a recently introduced language called Ptolemy [41] as it adapts ideas from both aspect-orientation and existing implicit invocation and has advantages over both of them. Since this language does not yet have a mature implementation, I do a theoretical analysis of the language. I specifically study the property of separate compilation. Separate compilation is very useful when more than one team wants to work on distinct parts of a program independently. The teams can compile their modules separately by just knowing the interface of the other teams. Other advantages include:

- program structuring - allows programmers to structure the program more modularly,

- incremental compilation

- reusability - better program structuring enables reusability.

In this chapter, I prove the properties of separate compilation for Ptolemy as defined by Cardelli [13] and also demonstrate the absence of such property in the popular model of aspects as in the AspectJ-like languages [28]. To prove this property for Ptolemy, I first introduce the Ptolemy language via an example in Section 3.1. I then introduce the language design of an object-oriented language, a simple aspect-oriented language and Ptolemy. Section 3.3 introduces the type system for my base object-oriented language and extensions for a simple AO language and Ptolemy. I also demonstrate that separate compilation for the simple AO language based on the type system does not hold. Then I introduce the framework for proving separate compilation as adapted from Cardelli's work [13] through the following sections. Section 3.4 extends the type system to include what Cardelli calls binding judgements (I rename it to module judgements to avoid confusion between this and the binding construct in Ptolemy) and signature judgements to model a module in Ptolemy (adapted from [13]). Section 3.6 for-

malizes the conversion from a module to a linkset. Finally, Section 3.7 proves the property of separate compilation for Ptolemy.

3.1 Introduction to Ptolemy

Ptolemy [41] is a programming language whose goals are similar to that of Aspect-oriented (AO) and implicit invocation (II) languages one of which is to enable the software engineers to separate conceptual concerns at the programming language level. But Ptolemy improves on both AO and II languages by taking the best of both worlds. Ptolemy supports typed events improving the idea of explicit announcement of events in II languages. It also supports quantified events like in AO languages by allowing the programmer to identify many events at once through the notion of quantified event types. Events are certain program points during the execution of a program. Ptolemy allows the programmer to announce events declaratively, similar to II languages but in a better way which allows around advices for events naturally. It also decouples the observers - entities which handle events from the subjects - entities which announce events, instead coupling them to event types thus allowing for separate teams to work independently on subject code and observer code. In general, Ptolemy has many advantages over both AO and II languages . I explain some of them briefly here, but they are explained in detail in [41].

Advantages of Ptolemy over AO languages are:

- No fragile pointcut problem - The quantified, typed events declared in Ptolemy does not depend on the syntactic properties of the program as events are declaratively announced.

- No quantification failure - Different kinds of join points according to AO languages can be announced as a single event type in Ptolemy, which allows quantifying over them. It also allows many more join point types than those that are supported by AO languages.

- User defined context information at events enables the user to expose unlimited context information for any given event type.

- Regular context information - Quantification of events based on event types allows regular context information for all join points of a given pointcut.

- Decoupled subject and observer code - Events are identified by event types denoted by **evtype** and are declaratively announced by subjects and handled by observers.

- Separate compilation - As I will show in Section 3.7, any given module can be compiled separately since all dependencies can be identified during type checking of the module.

Advantages of Ptolemy over II languages are:

- Decoupled subjects and observers - Both subjects and observers are coupled to the event types which act as interface between the two. This ensures that subjects and observers are independent of each other.

- Quantification - coupling the observers and subjects independently to event types allows for quantified events.

- *around* advice - handler chain is invoked at the beginning of an event and the handler methods control whether the original code is executed before, after or around the advice.

3.1.1 An Example in Ptolemy

To illustrate the key ideas in Ptolemy [41], let us consider a collection library shown in Figure 3.1. For simplicity assume that this library provides an interface `Collection` with only two methods: `add` to insert items in the collection and method `iter` to retrieve an iterator for the collection. The iterator interface also provides two methods: `hasNext` to check whether the next item exists and `next` to access the next item in the collection. The class `List` implements the interface `Collection`. It maintains the linked list data structure using a nested class `Node`. The method `add` for this class inserts the item at the end of the list. The method `iter` creates an instance of the anonymous nested class that allows iterations over the linked list.

An example requirement for such a collection could be to declare and announce addition and removal of items from the collection. Other components may be interested in such events, e.g. for implementing incremental functionality that relies on analyzing the increments. An example of such a concern for the linked list of integers is the requirement to keep track of the average of the integers in the list. Such an average would need to be updated when new integers are added and removed from the

```
1  /* Class Collection and Iterator elided */
2  class List extends Collection {
3  /* Class Node elided */
4  Node head = null, tail = null;
5  List add(int item) {
6   List list = this;
7   event ItemAdded {
8    if (head == null) {
9     head = new Node(item); tail = head;
10   } else {
11    tail.next = new Node(item); tail = tail.next;
12   }
13   this;
14   }
15   this;
16  } /* Method iter elided */ }
17  List evtype ItemAdded {
18  List list; int item;
19  }
20  class Average{
21  long count = 0; long average = 0;
22  Average(){ register(this);  }
23  List update(thunk List next, List list, int item){
24   invoke(next);
25   average = (average * count + item)/(++count);
26   list;
27  }
28  when ItemAdded do update;
29  }
```

Figure 3.1   An example collection with quantified, typed events: also shown is
the implementation of the average computation logic using quantified,
typed events.

list. Furthermore, such functionality may not be useful for all applications that use the linked list class,
thus it would be sensible to keep its implementation separate from that of the linked list class in order
to maximize reuse.

Typically such a requirement would be implemented using the observer design pattern [22], that is
directly supported in implicit invocation (II) languages [33, 36]. In II languages, *events* are seen as a
decoupling mechanism that is used to interface two sets of modules, so that they can be independent
of each other. Certain modules, often called subjects, dynamically and explicitly *announce* events.
Another set of modules, often called observers, can dynamically *register* methods, called *handlers*.
These handlers are invoked (implicitly) when events are announced. The subjects are thus independent
of the particular observers. Aspect-oriented (AO) languages [29] such as AspectJ [28] can also be used
to implement this requirement as demonstrated by Hannemann and Kiczales [24], Sakuraiet al. [44],
and Rajan [40]. However, both II and AO languages have several limitations described in detail in [41].

In II languages, observers remain coupled with subjects, no support for overriding is available, and specifying how each event is handled can grow in proportion to the number of objects from which implicit invocations are to be received [41]. AO languages have a fragile pointcut problem [47, 50], language-imposed limits on the types of events announced, and a limited interface for accessing contextual (or reflective) information about an event [41]. Quantified, typed events as described in [41] solve these problems.

In Ptolemy, **evtype** declarations allow programmers to declare named event types. An event type (**evtype**) declaration p has a return type, a name, and zero or more context variable declarations. These context declarations specify the types and names of reflective information communicated between announcement of events of type p and handler methods. These declarations are independent from the modules that announce or handle these events. The event types thus provide an interface that completely decouples subjects and observers. An example event type declaration is shown on lines 17–19 in Figure 3.1. The **evtype** ItemAdded declares that an event of this type makes two pieces of context available: the list that is being modified (list) and the item that is being inserted in the list (item).

Events are explicitly announced using **event** statements. These expressions enclose a body, which can be replaced by a handler. This functionality is akin to **around** advice in AO languages. The class List declares and announces an event of type ItemAdded using an event statement (lines 7–14). Arbitrary blocks can be declared as the body of the event statement. Event statements require the context variables to be bound in the lexical scope of their declaration. Event type ItemAdded declares two context variables: list and item. Thus the event statement in the body of the add method contains a local variable declaration that binds **this** to the name list (line 6). The name item is already bound to the formal parameter of the method add.

Finally, the names of **evtype** declarations can be utilized for quantification, which simplifies binding and avoids coupling observers with subjects. Bindings in Ptolemy associate a handler method to a set of events identified by an event type. The binding in Figure 3.1, line 28 says to run method update when events of type ItemAdded are announced. This allows selecting a number of event statements with just one succinct binding declaration without depending on the modules that announce

events.

3.2   Language Definitions

In this section, I will first describe the design of a small object-oriented language. I will then use this language as the basis for a simple, but representative, aspect-oriented (AO) extension that serves well to illustrate my ideas. I will also extend the object-oriented language with features from Ptolemy. Without loss of generality in all these language definitions I assume that classes have unique names, and all fields and methods are unique in a class. In the AO extension, in addition I also assume that the special (method-like) advice constructs are unique and aspects have unique names. In Ptolemy also, I assume that bindings are unique and that event types have unique names.

prog ::= *decl* e*
decl ::= **class** *c* **extends** *d* { *field* meth** }
field ::= *t f* ;
meth ::= *t m* (*form**) { e }
form ::= *t var*
t ::= *c*
e ::= **new** *c* () | *var* | **null** | *e.m* (*e**) | *e . f* | *e . f = e*
      | **cast** *c e* | *form = e* ; *e* | *e* ; *e*

$$
\begin{array}{rcl}
\textbf{where} & & \\
c,\ d & \in & \mathcal{C},\ \text{a set of class names} \\
f & \in & \mathcal{F},\ \text{a set of field names} \\
m & \in & \mathcal{M},\ \text{a set of method names} \\
var & \in & \{\textbf{this}\} \cup \mathcal{V}, \mathcal{V}\ \text{is} \\
& & \text{a set of variable names}
\end{array}
$$

Figure 3.2   Abstract Syntax of the Object-oriented Base Language

3.2.1   Abstract Syntax of the Object-oriented Language

The design of the object-oriented part is inspired from several languages including MiniMAO$_0$ [14, 15], Featherweight Java [26], and Classic Java [21]. It supports classes, objects, inheritance, and subtyping. However, it does not support method overloading, **super**, interfaces, exception handling, built-in value types, privacy modifiers, or abstract methods.

A program consists of a set of classes followed by an expression which is the "entry" point of the

```
decl ::= ... | aspect a { field* adv* }
adv ::= t around ( form* ) : pcd { e }
pcd ::= execution(re(c) re(t₀) re(m)(re(t̄)))
    | call (re(c) re(t₀) re(m)(re(t̄)))
e ::= ... | proceed ( e* )
```

where $a \in \mathcal{A}$, a set of aspect names

Figure 3.3   Abstract Syntax of Aspect-oriented Extension

program. A class inherits from exactly one other class. Furthermore, in the rest of the description I assume that this inheritance hierarchy does not contain any cycles. A class can have zero or more fields and methods. A method consists of a method header and a body (an expression). This language is purely object-oriented and hence all types are class types. An expression can be an instantiation, variable access, null, method invocation, field access, field set, cast, or a sequence of such expressions. The syntax of this language is shown in Figure 3.2

3.2.2   Aspect-oriented Extension

I extend my simple object-oriented language to support aspect-oriented features in the style of AspectJ-like languages [28]. This extension is shown in Figure 3.3. Any term not defined in this figure is assumed to be the same as defined in Figure 3.2. A term redefined as *term* ::= . . . | *term'* is assumed to be the combination of the definition of *term* in Figure 3.2 and *term'*. In this language, a program consists of a set of class or aspect declarations. I do not support inheritance among aspects. An aspect consists of a set of fields and advices. I only support **around** advices. However, both **before** and **after** can be supported by **around** advice. An advice specifies the functionality for the join point shadows captured by a pointcut. I support only two basic pointcuts, namely **execution** and **call** pointcuts. I support the proceed call in an **around** advice in a way that is similar to that of AspectJ. However, I do not support join point shadows in aspects and I do not support advice chaining. Other features of my object-oriented language are all part of this aspect-oriented language as well.

To compare the Ptolemy example with an equivalent program in a typical AO language, in Fig-

```
1 /* Class Collection and Iterator elided */
2 class List extends Collection {
3 /* Class Node elided */
4 Node head = null, tail = null;
5 List add(int item) {
6   if (head == null) {
7     head = new Node(item); tail = head;
8   } else {
9     tail.next = new Node(item); tail = tail.next;
10   }
11   this;
12 } /* Method iter elided */ }
13 aspect Average{
14 long count = 0;
15 List around(int item):List+ List+ add(int item) {
16   List list = proceed(item);
17   average = (average * count + item)/(++count);
18   list;
19 }
20 }
```

Figure 3.4 The collection example in our basic OO language: also shown is the
implementation of the average computation logic using an aspect

decl ::= **class** *c* **extends** *d* { *field* meth* binding** }
    | *t* **evtype** *p* { *form** }
t ::= ... | **thunk** *c*
e ::= ... | **register** (*e*) | **event** *p* { *e* } | **invoke** (*e*)
binding ::= **when** *p* **do** *m* ;

where $p \in \mathcal{P}$, a set of evtype names

Figure 3.5 Abstract Syntax of Ptolemy's Features, from Rajan and Leavens's
work [41, Fig. 2]

ure 3.4 I show the same example in my basic AO language in. In this case, I can see that the base code programmer can program the object-oriented part completely oblivious to the existence of aspects in the system. This allows for accommodating unanticipated evolution. Here, the object-oriented classes can be compiled without the aspects.   Thus, separate compilation holds true for classes as in a typical object-oriented language. However, compilation of an aspect needs to check all the applicable joinpoint shadows in all classes to match the pointcuts. Thus, aspects may not be compiled separately.

3.2.3 Ptolemy

The Ptolemy language [41] extends my simple object-oriented language. It includes a novel event model. The new features of Ptolemy are an event type declaration and event expression represented in the syntax as **evtype** and **event** respectively.

Ptolemy has the ability to replace all events in a specified set with a call to a handler method. A handler takes an event closure as its first argument. An *event closure* [43] has the code needed to run the applicable handlers and the original event's code. It is invoked by the expression **invoke**.

The language allows any class to register handlers to events. It allows a handler to be registered with a set of events that are identified using event types by a declarative construct called *binding*. The expression **register** activates an instance of a class to receive events bound to the handlers in the class through the *bindings*.

The syntax of the language is shown in Figure 3.5. A program consists of a sequence of top-level declarations followed by a main expression which is the starting point of the program. A top-level declaration could be either a class or an event type (**evtype**). A class can consist of a sequence of field declarations, followed by a sequence of method declarations, followed by a list of bindings, all of which are optional. A binding binds an event type, to a method which acts as a handler. An **evtype** contains a sequence of field declarations (again optional) which represents the context of an event of that type. It also has a return type which should be the return type of all events of that type. A type t could be either a class type or an event closure type (e.g. **thunk** $List$ in Figure 3.1 where $List$ is the return type).

Along with some usual OO expressions, three event-based expressions are supported by Ptolemy which are **register**, **event** and **invoke**. The **register** expression registers the argument object as an active object so that it can receive the events that it is bound to (by the bindings). The **event** expression declares a block of code as an event so that when the control comes to the event during execution, the correct handler is called along with the context information and the event closure. The **invoke** expression runs the event closure identified by the argument.

3.3   Separate Compilation as Type Checking

Cardelli in his seminal work on separate compilation [13] defines separate compilation as "separate typechecking and separate code generation of program fragments". He also argues that for all practical purposes, separate typechecking is enough to prove separate compilation. Hence, I follow him to prove separate type checking in Ptolemy to prove separate compilation.

In order to analyze separate typechecking properties, in this section, I show the typechecking rules for the base object-oriented language, the aspect-oriented language that I introduced and for Ptolemy.

A type environment (Γ) is defined as a partial finite function from names to types as shown in Figure 3.6. I modify it slightly (similar to [3]) to include mappings from class names and event names to their types along with the usual variable names to their types. The issue of a variable name clashing with a class name in an environment can be resolved by preceding the type checking phase with a name mangling phase which mangles the names of the variables such that there are no clashes with the class names in the type environment. It can also be resolved by maintaining a flag along with each name which indicates if it is the name of a top-level declaration or if it is a regular name.

Figure 3.6 also refers to field, method, advice and binding types which are defined in Figure 3.7. $\bar{t}$ represents a list of parameter types (often in a method). A field is represented by the type and name of the field. A method is represented by the return type, name and parameter list of the method. An advice is represented by a tuple specifying the advice type (by the flag 'a'), the return type, target type, bound parameters and the pointcut that it is advising. A binding is represented by a tuple specifying the binding type (by the flag 'b'), the event type that is being bound and the method that the event type is being bound to.

Figure 3.8 defines the short-hand notations used in the rest of this paper to represent the bodies declaring classes, aspects and event types of the basic object-oriented languages, basic aspect-oriented language and Ptolemy respectively.

3.3.1   Type checking rules for the base OO language

Figure 3.9 defines typing rules for the base object-oriented language. The domain of an environment which is names of an environment is returned by the function dom. The rule (PROGRAM) defines

$\theta ::=$ "type attributes"
   | OK   "program/top-level decl."
   | OK in c "method, binding"
   | β "program/top-level type"
   | $field$ "field type"
   | $mdecl$ "method type"
   | $bdecl$ "binding type"
   | $adecl$ "advice type"
   | **var** t "var/formal/field"
   | **exp** t "expression"
$\Gamma ::= \{I : \theta_I\}_{I \in K},$ "type environments"
       where K is finite, $K \subseteq (\mathcal{L} \cup \{\textbf{this}\} \cup \mathcal{V})$
$\Pi ::= \{x : t\}$ "method parameter environment"
$\beta ::= \langle \kappa, c, c', field^*, mdecl^* \rangle$ "class type"
   | $\langle \epsilon, p, field^*, c \rangle$ "event type"
   | $\langle \alpha, a, field^*, adecl^* \rangle$ "aspect type"

Figure 3.6   Type attributes

$\bar{t} ::= \Lambda | t_1 \ldots t_n$
$field ::= t\ f$
$field^* ::= \Lambda | field_1 \ldots field_n$
$mdecl ::= t\ m(\bar{t})$
$mdecl^* ::= \Lambda | mdecl_1 \ldots mdecl_n$
$adecl ::= \langle a, t_0, c, t_1\ x_1, \ldots, t_n\ x_n, pcd \rangle$
$adecl^* ::= \Lambda | adecl_1 \ldots adecl_n$
$bdecl ::= \langle b, p, m \rangle$
$bdecl^* ::= \Lambda | bdecl_1 \ldots bdecl_n$

Figure 3.7   Definitions of signatures

the typechecking of a program. It checks the top-level declarations followed by the main expression. The rule (CLASS) defines the typechecking of a class declaration. It first checks that the class extends a valid class. Then it checks that all fields and methods are typed correctly.

The rule (NAME-HAS-TYPE) states that the name of a class is associated with a type if the class declaration is well-formed.

The rule (METHOD) states that a method declaration is valid if the body is valid and returns a type that is a subtype of the declared return type and all argument types exist in Γ. If the arguments are used in the body of the method, they will be typechecked while typechecking the body. If they are not used in the body, then it does not break the soundness of the type system. The judgement $\Gamma \vdash T \preccurlyeq T'$ is valid whenever T is a subtype of T' in the environment Γ as defined in the Figure 3.13.

$c\sharp ::= \textbf{class } c \text{ extends } d\{field^*\ meth^*\}$
$p\sharp ::= t\ \textbf{evtype } p\{field^*\}$
$cp\sharp ::= \textbf{class } c \text{ extends } c'\{field^*\ meth^*\ bdecl^*\}$
$\quad | \ t\ \textbf{evtype } p\{field^*\}$

Figure 3.8   Short notations

The rule (FIELD) defines typechecking of a field signature. A field signature is valid if the type of the field is defined in Γ.

The rest of the rules in the object-oriented part define the typechecking of expressions.

The rule (NEW) defines that an instance creation expression **new** $c()$ is well-typed and has type c in Γ, Π provided that c exists in Γ. The environment Π represents the local environment within the method in which the expression is found.

The rule (CALL) checks the method call expression. It first checks the types of the receiver expression and all the argument expressions. Then the most applicable method based on rules of overriding is determined using the function $methRes$ defined in Figure 3.12.

The rule (INIT) defines typechecking of the variable declaration and initialization expression. It checks that the type of the expression to which the variable is initialized is a subtype of the declared type of the variable and that the declared type of the variable is present in Γ, Π. It also requires that the expression is followed by another kind of expression to ensure that variable declarations are at the beginning of a block similar to the *C* language.

The rule (GET) defines typechecking of the expression to get a variable from an object by checking that the type of the target object either declares the field itself or inherits from one of its parent classes. Similarly, the rule (SET) checks the type of the target object either declares or inherits the field and also that the expression to which the field is initialized is a subtype of the declared type of the field. The expression $c \triangleleft t\ f$ means that the class c either declares the field f of type t or inherits it from one of it's parent classes. Figure 3.13 defines this expression.

The last rule for the base object-oriented language (EXPR. SEQUENCE) defines type checking of sequence of expressions.

(PROGRAM)

$$\frac{(\forall i \in \{1..n\} :: \Gamma \vdash decl_i : \text{OK}) \qquad \Gamma \vdash e : \textbf{exp}\ t}{\vdash decl_1 \ldots decl_n\ e : \textbf{prog}\ t}$$

(CLASS)

$$\frac{isClass(d) \qquad (\forall i \in \{1..m\} :: \ \vdash field_i : \text{OK in } c) \qquad (\forall j \in \{1..n\} :: \ \vdash meth_j : \text{OK in } c)}{\Gamma \vdash \textbf{class}\ c\ \textbf{extends}\ d\{field_1 \ldots field_m\ meth_1 \ldots meth_n\} : \text{OK}}$$

(NAME-HAS-TYPE)

$$\frac{\Gamma \vdash \textbf{class}\ c\ \textbf{extends}\ d\{field_1 \ldots field_m\ meth_1 \ldots meth_n\} : \text{OK}}{\Gamma \vdash c : \langle \kappa, c, d, field_1 \ldots field_m, mdecl_1 \ldots mdecl_n \rangle}$$

(METHOD)

$$\frac{\begin{array}{cc} \Gamma; \{x_i : \textbf{var}\ t_i | 0 \le i \le n, this : \textbf{var}\ c\} \vdash e : t & \Gamma \vdash t \preccurlyeq t_0 \end{array}}{(\forall i \in \{0 \ldots n\} :: t_i \in dom(\Gamma)) \quad this : \textbf{var}\ c \quad (\textbf{class}\ c\ \textbf{extends}\ d\ldots) \in CT \quad override(m, d, t_1, \times \ldots \times t_n \to t_0)}$$
$$t_0\ m(t_1\ x_1, \ldots, t_n\ x_n)\{e; \} : \text{OK in } c$$

(FIELD)

$$\frac{t \in dom(\Gamma) \qquad this : \textbf{var}\ c \qquad isClass(c)}{\Gamma \vdash t f : \text{OK in } c}$$

(NEW)

$$\frac{c \in dom(\Gamma)}{\Gamma; \Pi \vdash \textbf{new}\ c() : \textbf{exp}\ c}$$

(CALL)

$$\frac{\Gamma; \Pi \vdash e_0 : \textbf{exp}\ c}{(\forall i \in \{1 \ldots n\} :: \Gamma; \Pi \vdash e_i : \textbf{exp}\ t_i) \qquad (\forall i \in \{1 \ldots n\} :: t_i \preccurlyeq t_i') \qquad methRes(\Gamma, c, m) = \langle c', t', t_1' \ldots t_n' \rangle}$$
$$\Gamma; \Pi \vdash e_0.m(e_1, \ldots, e_n) : \textbf{exp}\ t'$$

(INIT)

$$\frac{\Gamma; \Pi \vdash e_1 : \textbf{exp}\ t' \qquad t' \preccurlyeq t \qquad t \in dom(\Gamma, \Pi) \qquad \Gamma; \Pi \vdash e_2 : \textbf{exp}\ t''}{\Gamma; \Pi \vdash t\ var = e_1; e_2 : \textbf{exp}\ t''}$$

(GET)

$$\frac{\Pi \vdash e : \textbf{exp}\ c \qquad c \triangleleft t\ f}{\Pi \vdash e.f : \textbf{exp}\ t}$$

(SET)

$$\frac{\Gamma; \Pi \vdash e : \textbf{exp}\ c \qquad c \triangleleft t\ f \qquad \Gamma; \Pi \vdash e' : \textbf{exp}\ t' \qquad t' \preccurlyeq t}{\Gamma; \Pi \vdash e.f = e' : \textbf{exp}\ t'}$$

(EXPR. SEQUENCE)

$$\frac{\Gamma; \Pi \vdash e_1 : \textbf{exp}\ t_1 \qquad \Gamma; \Pi \vdash e_2 : \textbf{exp}\ t_2}{\Gamma; \Pi \vdash e_1; e_2 : \textbf{exp}\ t_2}$$

Figure 3.9   Type-checking rules for the base OO language

3.3.2   Type checking rules for the basic AO language

The basic AO language that we study here extends the base OO language I introduced in the previous section (similar to AspectJ being a superset of Java). Hence all the type checking rules introduced there remain intact. The additional type checking rules for the language are shown in Figure 3.10.

The rule (ASPECT) defines the type checking of an aspect. It checks that all fields and methods are typed correctly.

The rule (ASPECTNAME-HAS-TYPE) checks that the name of an aspect is associated with a type if the aspect declaration is well-formed.

The rules (EXECUTION PCD) and (CALL PCD) define type checking of a PCD in the language. It

(ASPECT)

$$\frac{(\forall i \in \{1..m\} :: \; \vdash field_i : OK) \qquad (\forall i \in \{1..n\} :: \; \vdash adv_i : OK)}{\Gamma \vdash \textbf{aspect} \; a\{field_1 \ldots field_m \; adv_1 \ldots adv_n\} : OK}$$

(ASPECTNAME-HAS-TYPE)

$$\frac{\Gamma \vdash \textbf{aspect} \; a\{field_1 \ldots field_m \; adv_1 \ldots adv_n\} : OK}{\Gamma \vdash a : \langle \alpha, a, field_1 \ldots field_m \; adecl_1 \ldots adecl_n \rangle}$$

(EXECUTION PCD)

$$\frac{(\forall i \in \{0 \ldots n\} :: \textbf{re}(c), \textbf{re}(t_i) \subset \mathcal{RE})}{\Gamma; \Pi \vdash \textbf{execution}(\textbf{re}(c) \; \textbf{re}(t_0) \; \textbf{re}(m)(\textbf{re}(t_1) \; x_1 \ldots \textbf{re}(t_n) \; x_n)) : \quad \langle p, \textbf{re}(c), \textbf{re}(m), \textbf{re}(t_0), \textbf{re}(\overline{t}) \rangle}$$

(CALL PCD)

$$\frac{(\forall i \in \{0 \ldots n\} :: \textbf{re}(c), \textbf{re}(t_i) \subset \mathcal{RE})}{\Gamma; \Pi \vdash \textbf{call} \; (\textbf{re}(c) \; \textbf{re}(t_0) \; \textbf{re}(m)(\textbf{re}(t_1) \; x_1 \ldots \textbf{re}(t_n) \; x_n)) : \quad \langle p, \textbf{re}(c), \textbf{re}(m), \textbf{re}(t_0), \textbf{re}(\overline{t}) \rangle}$$

(ADVICE)

$$\frac{\Gamma \vdash pcd : \langle p, \textbf{re}(c), \textbf{re}(m), \textbf{re}(t_0'), \textbf{re}(t_1') \ldots \textbf{re}(t_n') \rangle \qquad (\forall i \in \{0 \ldots n\} :: matches\_RE(\textbf{re}(t_i'), t_i) = true)}{\Gamma \vdash match(pcd) = MJP \quad (\forall i \in \{1 \ldots n\} :: \Gamma; x_i : t_i \vdash e : t) \qquad \Gamma \vdash t \preccurlyeq t_0 \qquad \Gamma \vdash this : \textbf{var} \; a \qquad isAspect(a)}{\Gamma \vdash t_0 \; \textbf{around} \; (t_1 \; x_1, \ldots, t_n \; x_n)" : "pcd\{e\} : OK \; in \; a \qquad \Gamma; \Pi \vdash proceed : (c \times t_1 \times \ldots \times t_n \rightarrow t_0)}$$

(CHECK PROCEED)

$$\frac{\forall i \in 1 \ldots n :: \Gamma; \Pi \vdash e_i : t_i' \qquad \Gamma; \Pi \vdash proceed : (c \times t_1 \times \ldots \times t_n \rightarrow t_0) \qquad \forall i \in 1 \ldots n :: \Gamma; \Pi \vdash t_i' \preccurlyeq t_i}{\Gamma; \Pi \vdash proceed(e_1, \ldots, e_n) : \textbf{exp} \; t_0}$$

Figure 3.10   Additional type-checking rules for basic AO language

supports only execution and call pointcut without supporting many other pointcuts like **this**, cflow, etc. that are supported in many AspectJ-like languages. It also uses a slightly different form of method signature than followed in AspectJ. It just checks that the regular expressions used in specifying the method signatures are valid regular expressions. I do not define the exact definition of the set $\mathcal{RE}$, but it can be assumed to be the same as in AspectJ or any such language.

The rule (ADVICE) defines the type checking of an advice. Observe that the language supports only around advices. It checks that the argument list for the around advice matches the argument list mentioned in the PCD. The around advice need not declare all the argument types defined in the PCD. My language supports **proceed** with similar semantics to that supported in AspectJ, where the arguments of the proceed call are the arguments declared by the advice and the return type is the return type of the advice. Type checking uses a variable called "**this**" which just indicates the type being checked. In this case, it just checks that the advice it is checking is in an aspect A. The type checking also includes matching all joinpoint shadows in the system with the pointcut that it is advising. This breaks the property of separate compilation as I will explain in Section 3.3.4.

The rule (CHECK PROCEED) defines the type checking of proceed call within an advice. The parameters of the proceed call are type checked to be sub types of the parameters declared by the advice and the return type is type checked to be the sub type of the declared return type of the advice.

3.3.3 Type checking rules for Ptolemy

The Ptolemy language extends from the base OO language I introduced in Section 3.3.1. Hence most of the type checking rules introduced there remain intact. However, the typechecking of a rule, a top-level declaration changes. These modified and additional type checking rules for the language are shown in Figure 3.10.

(CLASS)

$$\frac{isClass(d)}{(\forall i \in \{1..l\} :: \ \vdash field_i \ : \ OK \ in \ c) \qquad (\forall j \in \{1..m\} :: \ \vdash meth_j \ : \ OK \ in \ c) \qquad (\forall k \in \{1..n\} :: \ \vdash binding_k \ : \ OK \ in \ c)}{\Gamma \vdash \textbf{class} \ c \ \textbf{extends} \ d\{field_1 \dots field_l \ meth_1 \dots meth_m \qquad binding_1 \dots binding_n \ \} : OK}$$

(NAME-HAS-TYPE)

$$\frac{\Gamma \vdash \textbf{class} \ c \ \textbf{extends} \ d\{field_1 \dots field_l \ meth_1 \dots meth_m \qquad binding_1 \dots binding_n \} \ : \ OK}{\Gamma \vdash c : \langle \kappa, c, d, field_1 \dots field_l, mdecl_1 \dots mdecl_m \qquad bdecl_1 \dots bdecl_n \rangle}$$

(EVENT)

$$\frac{(\forall i \in \{1..l\} :: \ \vdash field_i \ : \ OK \ in \ p) \qquad isClass(c)}{\Gamma \vdash c \ \textbf{evtype} \ p\{field_1 \dots field_l\} : OK}$$

(EVENTNAME-HAS-TYPE)

$$\frac{\Gamma \vdash c \ \textbf{evtype} \ p\{field_1 \dots field_n\} \ : \ OK}{\Gamma \vdash p : \langle \epsilon, \ p, \ field_1 \dots field_n, \ c\rangle}$$

(CONTEXT FIELD)

$$\frac{t \in dom(\Gamma) \qquad this : \textbf{var} \ p \qquad isEVType(p)}{\Gamma \vdash tf \ : OK \ in \ p}$$

(EVENT EXPR.)

$$\frac{\Gamma \vdash p : \langle \epsilon, p, field_1 \dots field_l, c\rangle}{\Gamma; \Pi \vdash e : t' \qquad t' \preccurlyeq c \qquad (\forall 1 \le i \le n :: field_i = t_i \ f_i \ and \ \Gamma; \Pi \vdash f_i : t_i' \ and \ t_i' \preccurlyeq t_i)}{\Gamma; \Pi \vdash \textbf{event} \ p\{e\} : \textbf{exp} \ t}$$

(REGISTER)

$$\frac{\Gamma; \Pi \vdash e : \textbf{exp} \ t}{\Gamma; \Pi \vdash \textbf{register}(e) : \textbf{exp} \ t}$$

(BINDING SIGNATURE)

$$\frac{\Gamma \vdash p : \langle \epsilon, p, field_1 \dots field_l, c_1\rangle \qquad \Gamma \vdash c \vartriangleleft c_1' \ m(t_1, \dots, t_n)}{(c_1' \ m(t_1 \ var_1, \dots, t_n \ var_n)\{e\}) = \texttt{methRes}(c, m) \qquad n \ge 1 \qquad (\forall 1 \le i \le l :: field_i = t_{i+1}' \ var_{i+1})}{l = n - 1 \qquad (\forall 2 \le i \le n :: t_i \le t_i' \qquad t_1 = \textbf{thunk} \ c_1 \qquad c_1' \preccurlyeq c_1 \qquad this : \textbf{var} \ c \qquad isClass(c)}{\Gamma \vdash \textbf{when} \ p \ \textbf{do} \ m : OK \ in \ c}$$

(INVOKE)

$$\frac{\Gamma; \Pi \vdash e : (\textbf{thunk} \ t)}{\Gamma; \Pi \vdash \textbf{invoke}(e) : \textbf{exp} \ t}$$

Figure 3.11   Additional type-checking rules for Ptolemy, adapted from the work of Rajan and Leavens [41].

The rules (CLASS) and (CLASS-HAS-TYPE) remain close to its original definition except that a new

construct called binding is added.

The rule (EVENT) defines typechecking of an event type. It type checks all the field signatures and checks that the return type c is a valid class.

The rule (EVENTNAME-HAS-TYPE) checks that the name of an event type is associated with a type if the class declaration is well-formed.

The rule (CONTEXT FIELD) defines typechecking of a context field signature. A context field signature is valid if the type of the field is defined in Γ.

The rule (EVENT EXPR.) defines typechecking of an event expression by checking that the name of the event is defined in Γ, the return type of the event body is a subtype of the declared return type of the event and all the fields in the event type are defined in Γ, Π.

The rule (REGISTER) says that type of a register expression is the type of the expression being registered.

The rule (BINDING SIGNATURE) defines the typechecking of a binding. It checks that the event type is defined in Γ, the advising method m is either inherited or defined in the enclosing class, the first parameter of the method exists and is an event closure of the type which is a subtype of the returning type of the event and that all the context variables of the event are defined in the scope of the event. The function $methodBody$ returns the body of the method passed as a parameter contained in the class which is another parameter.

The rule (EVENT TYPE) says that the name of an event type is associated with a type if the class declaration is well-formed.

The rule (INVOKE) checks the type of an invoke expression to be of an event closure type.

$$\frac{\Gamma \vdash c : \langle c, d, \_, mdecl_1 \ldots mdecl_k \rangle \qquad \exists i \in \{1 \ldots k\} :: mdecl_i = t\, m(t_1 \ldots t_n)}{methRes(\Gamma, c, m) = \langle c, t, t_1 \ldots t_n \rangle}$$

$$\frac{\Gamma \vdash c : \langle c, d, \_, mdecl_1 \ldots mdecl_k \rangle \qquad \nexists i \in \{1 \ldots k\} :: mdecl_i = t\, m(t_1 \ldots t_n) \qquad methRes(\Gamma, d, m) = \langle d', t, t_1 \ldots t_n \rangle}{methRes(\Gamma, c, m) = \langle d', t, t_1 \ldots t_n \rangle}$$

Figure 3.12 Definition of $methRes$, inspired by [3]

$$\frac{\Gamma \vdash c}{\Gamma \vdash c \preccurlyeq c} \qquad \frac{\Gamma \vdash c :: \_}{\Gamma \vdash c \preccurlyeq c'} \quad \Gamma(c) = \langle \kappa, c, c', \_, \_, \_\rangle \qquad \frac{\Gamma \vdash c \preccurlyeq c' \quad \Gamma \vdash c' \preccurlyeq c''}{\Gamma \vdash c \preccurlyeq c''} \qquad \frac{\forall i \in 1 \ldots n :: \Gamma \vdash t_i \preccurlyeq t_i'}{\Gamma \vdash t_1 \ldots t_n \preccurlyeq t_1' \ldots t_n'}$$

$$\frac{field \in fieldsOf(c)}{\Gamma \vdash c \vartriangleleft field} \qquad \frac{\Gamma \vdash c \preccurlyeq c' \quad field \in fieldsOf(c')}{\Gamma \vdash c \vartriangleleft field} \qquad \frac{mdecl \in mdeclsOf(c)}{\Gamma \vdash c \vartriangleleft mdecl}$$

$$\frac{\Gamma \vdash c \preccurlyeq c' \quad mdecl \in mdeclsOf(c')}{\Gamma \vdash c \vartriangleleft field}$$

Figure 3.13   Implementation and Widening, based on [3]

MATCH CLASSES
$$\frac{\Gamma \vdash dom(CT) = \{c_1, c_2, \ldots, c_n\} \qquad (\forall i \in 1 \ldots n :: \Gamma \vdash match(pcd, c_i) = MJP_i)}{\Gamma \vdash match(pcd) = MJP_1 \cup MJP_2 \cup \ldots \cup MJP_n}$$

MATCH METHODS
$$\frac{\Gamma \vdash c : \langle \kappa, c, d, \_, mdecl* \rangle \qquad mdecl* = mdecl_1 \, mdecl_2 \ldots mdecl_n \qquad (\forall i \in 1 \ldots n :: \Gamma \vdash match(pcd, c, mdecl_i) = mjp_i) \qquad (\forall i \in 1 \ldots n :: \Gamma \vdash match(pcd, d, mdecl_i) = MJP)}{\Gamma \vdash match(pcd, c) = MJP \cup mjp_1 \cup mjp_2 \cup \ldots \cup mjp_n}$$

MATCH METHOD
$$\frac{\Gamma \vdash c \vartriangleleft mdecl \qquad mdecl = t_0 \, m(t_1, \ldots, t_n) \qquad \Gamma \vdash pcd : \langle p, c^{RE}, m^{RE}, t_0^{RE}, \bar{t}^{RE} \rangle \qquad matches\_RE(c^{RE}, c) = true \qquad matches\_RE(t_0^{RE}, t_0) = true \qquad matches\_RE(\bar{t}^{RE}, t_1, \ldots, t_n) = true}{\Gamma \vdash match(pcd, c, mdecl) = \{c \, mdecl\}}$$

Figure 3.14   Definition of the auxiliary function match, inspired by [14, 52]

3.3.4   Absence of separate compilation for AspectJ-like languages

As we can see from the rule (ADVICE), type checking of an advice involves matching the pointcut that is being advised to all the joinpoint shadows in the system to find the applicable ones. This is done by the function $match$ which is defined in Figure 3.14. We can see from the definition that all the classes in the system are checked to see if they have any joinpoint shadows that match the pointcut. To do this, it requires knowledge of the entire configuration of the system and not just the modules that the aspect depends on. This breaks the property of separate type checking. Hence by Cardelli's definition of separate compilation, this language breaks the property of separate compilation. AspectJ-like languages have all the features of this language along with many more properties which break separate compilation. From hereon, I will focus on proving the separate compilation property for Ptolemy.

3.4 Modules for Ptolemy

In Ptolemy, there are three kinds of top-level declarations that are interesting from the point of view of separate compilation.

- Classes which announce events, i.e. subjects

- Classes which handle events, i.e. observers

- Event types which act as interface between these two

Classes which neither announce nor handle events can be grouped along with the classes which announce events for the sake of this discussion. Luca Cardelli in his seminal work on separate compilation [13] defines a program fragment as "any syntactically well-formed program term, possibly containing free variables". By this definition, all the above top-level declarations are program fragments. They all refer to the fields and methods of other classes except for event types which are simply interfaces between the subjects and observers. Cardelli also defines separate compilation as "separate typechecking and separate code generation of program fragments". Like Cardelli, I avoid issues of code generation by always working at the source-language level as in [13].

I use Cardelli's definition of a module as a collection of program fragments. Now to prove that separate compilation holds in Ptolemy, I have to prove the separate compilation property of any given module. I use Cardelli's framework for proving separate compilation making adjustments wherever required to adapt it to the requirements of Ptolemy.

Cardelli introduces two judgements called binding judgement and signature judgement to help in modularizing. I adapt them to Ptolemy by renaming Cardelli's binding judgement to module judgement to avoid confusion with the binding that the language has. Similar to a binding in [13], a module is a tuple of definitions and a signature is essentially a tuple of declarations. A module judgement represents modules while a signature judgement represents the export list for modules. Each item of the export list represents the visible type of the module it represents. For e.g. $cp_1 : s_1$ says that the module cp_1 is of the type s_1. Each item of a module (which can either be a class or an event type) not only identifies the visible type, but also the definition of that item. For e.g. $cp_1 : s_1 \doteq cp_1\sharp$ says that the module cp_1 is of type s_1 and is defined by $cp_1\sharp$.

So a module is of the form $cp_1 : s_1 \doteq cp_1\sharp, \ldots, cp_n : s_n \doteq cp_n\sharp$ and a signature is of the form $cp_1 : s_1, \ldots, cp_n : s_n$ where s_i is of the form $\langle \kappa, cp_i, cp_i', field^i*, mdecl^i*, bdecl^i* \rangle$ or $\langle \epsilon, cp_i, field^i*, c_i \rangle$. A signature judgement is of the form $\Gamma \vdash S$ (signature S is well-formed in Γ) and a module judgement is of the form $\Gamma \vdash M \therefore S$ (module M has signature S in Γ). The $\therefore$ symbol in module judgements is similar to Cardelli's work.

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \phi} \qquad \frac{\Gamma, cp : s \vdash S \quad \Gamma, S \vdash cp : s}{\Gamma \vdash cp : s, S} \qquad \frac{\Gamma \vdash \diamond}{\Gamma \vdash \phi \therefore \phi} \qquad \frac{\Gamma, cp : s \vdash M \therefore S \quad \Gamma, S \vdash cp : s \doteq cp\sharp}{\Gamma \vdash (cp : s \doteq cp\sharp, M) \therefore (cp : s, S)}$$

Figure 3.15   Signatures and Modules for Ptolemy

Figure 3.15 lists the rules that define signatures and modules for Ptolemy. In $\Gamma \vdash M \therefore S$ every component of M is matched by the corresponding component in S. Notice that the second and fourth rules ensure that $cp : s$ is compatible to S and vice versa. This ensures that the valid signature and module judgements satisfy the equivalent of inter-checking property for linksets.

To illustrate the module and signature judgements, I show some signature and module judgements with respect to a few possible modules from the example in Figure 3.1.

Examples of Signature Judgement

For a module consisting of the class `List`:

```
(Object : ⟨κ, Object,⊤, φ, φ, φ⟩,
Collection : ⟨κ, Collection, Object, _, _, _⟩
Iterator : ⟨κ, Iterator, Object, _, _, _⟩
Integer : ⟨κ, Integer, Object, _, _, _⟩
Node : ⟨κ, Node, Object, _, _, _⟩
List : ⟨κ, List, Collection, Node head, Node y,
List add(Integer), Iterator iter(), φ⟩,
ItemAdded : ⟨ε, ItemAdded, List list, Integer item, List ⟩)
⊢
(List : ⟨κ, List, Collection, Node head, Node y,
List add(Integer), Iterator iter(), φ⟩ ≐ FElement♯)
```

For a module consisting of the class `Average`:

```
(Object : ⟨κ, Object, ⊤, φ, φ, φ ⟩
```

```
Integer : ⟨κ, Integer, Object, _, _, _⟩
```

```
Long : ⟨κ, Long, Object, _, _, _⟩
```

```
List : ⟨κ, List, Collection, Node head, Node y,
```

```
ItemAdded : ⟨ε, ItemAdded, List list, Integer item, List ⟩)
```

⊢

```
(Average : ⟨κ, Average, Object, Long count, Long average, List
```

update(**thunk** List, List, Integer), ⟨$b, ItemAdded, update$⟩ ⟩ $\dot{=}$

Average♯)

Examples of Module Judgement

For a module consisting of the class `List`:

```
(Object : ⟨κ, Object, ⊤, φ, φ, φ⟩,
```

```
Collection : ⟨κ, Collection, Object, _, _, _⟩
```

```
Iterator : ⟨κ, Iterator, Object, _, _, _⟩
```

```
Integer : ⟨κ, Integer, Object, _, _, _⟩
```

```
Node : ⟨κ, Node, Object, _, _, _⟩
```

```
List : ⟨κ, List, Collection, Node head, Node y,
```

```
List add(Integer), Iterator iter(), φ⟩,
```

```
ItemAdded : ⟨ε, ItemAdded, List list, Integer item, List ⟩)
```

⊢

```
(List : ⟨κ, List, Collection, Node head, Node y,
```

List add(Integer), Iterator iter(), φ⟩ $\dot{=}$ FElement♯)

∴

```
(List : ⟨κ, List, Collection, Node head, Node y,
```

```
List add(Integer), Iterator iter(), φ⟩)
```

For a module consisting of the class `Average`:

```
(Object : ⟨κ, Object, ⊤, φ, φ, φ ⟩
```

```
Integer : ⟨κ, Integer, Object, _, _, _⟩
```

```
Long : ⟨κ, Long, Object, _, _, _⟩
```

```
List : ⟨κ, List, Collection, Node head, Node y,
```

```
ItemAdded : ⟨ε, ItemAdded, List list, Integer item, List ⟩)
```

⊢

```
(Average : ⟨κ, Average, Object, Long count, Long average, List
```

update(**thunk** List, List, Integer), ⟨$b, ItemAdded, update$⟩ ⟩ $\dot{=}$

Average♯)

∴

```
(Average : ⟨κ, Average, Object, Long count, Long average, List
```

update(**thunk** List, List, Integer), ⟨$b, ItemAdded, update$⟩ ⟩)

Now separate compilation of a module M can be seen as compilation of the module judgement $\Gamma \vdash M \therefore S$ provided Γ contains sufficient information about external fragments that any fragment in the module depends on.

Since I adapt the framework for separate compilation from [13], separate compilation maps module judgements $\Gamma \vdash M \therefore S$ into entities called linksets. Linksets and their properties relevant to separate compilation are introduced in the next section.

3.5   Linksets

The concept of linksets was introduced by Luca Cardelli as a configuration language to describe how a given set of program fragments should be linked. Informally, a linkset is a list of program fragments expressed as a collection of named judgements. The name of a judgement in a linkset is typically the name of the program fragment that the judgement represents. I use an adaptation of the definition of a linkset in this work. But the idea behind the concept remains the same.

$$\Gamma_0 \mid cp_1 \blacktriangleright \Gamma_1 \vdash \tau_1 \ldots cp_n \blacktriangleright \Gamma_n \vdash \tau_n$$

This is a linkset consisting of an environment Γ_0 and a collection of judgements $\Gamma_i \vdash \tau_i$ where τ_i is of the form **class** cp_i **extends** cp_i' $\{field^{i*}\, meth^{i*}\, binding^{i*}\} : \langle \kappa, cp_i, cp_i{}', field^{i*}, mdecl^{i*},$ $bdecl^{i*}\rangle$ or c_i **evtype** cp_i $\{field^{i*}\} : \langle \epsilon, cp_i, field^{i*}, c_i \rangle$. Each judgement is named by a label cp_i. The components $cp_i \blacktriangleright \Gamma_i \vdash \tau_i$ are called linkset fragments.

For example, consider the class List from the example in Figure 3.1. The linkset consisting of this class could be

```
(Object : ⟨κ, Object,⊤, φ, φ, φ⟩,
Collection : ⟨κ, Collection, Object, _, _, _⟩
Iterator : ⟨κ, Iterator, Object, _, _, _⟩
Integer : ⟨κ, Integer, Object, _, _, _⟩
Node : ⟨κ, Node, Object, _, _, _⟩
List : ⟨κ, List, Collection, Node head, Node y,
List add(Integer), Iterator iter(), φ⟩,
ItemAdded : ⟨ε, ItemAdded, List list, Integer item, List ⟩)
|
List ▶ List♯ : ⟨κ, List, Collection, Node head, Node y,
List add(Integer), Iterator iter(), φ⟩,
```

As in [13], the key ideas are that Γ_0 is the external interface of the entire linkset meaning that it is a shared environment of all fragments and hence $\Gamma_0, \Gamma_i \vdash \tau_i$ is a valid judgement. The names of the judgements are typically the names of the fragments that they represent and they must be unique. Also these names match the free variables of other fragments and thus determine the dependencies between the fragments.

For convenience, I list the following formal definitions of linksets and their related properties and some lemmas taken from Luca Cardelli. Some of them are modified slightly to suit Ptolemy.

Definition 3.5.1 *(Linkset structure)*

Consider the structure $L \equiv \Gamma_0 | cp_i \blacktriangleright \Gamma_i \vdash \tau_i^{i \in 1...n}$, where each τ_i has the shape **class** *cp_i* **extends** *cp_i' {field^i* mdecl^i* binding^i*} : $\langle \kappa,\ cp_i,\ cp_i$',\ field^i*,\ mdecl^i*,\ bdecl^i* \rangle$ or c_i* **evtype** *cp_i {field^i*} :$\langle \epsilon, cp_i, field^i$*, $c_i \rangle$. Each judgement is named by a label cp_i.*

Let $imp(L) \doteq dom(\Gamma_0)$ be the imported names of L.

Let $exp(L) \doteq \{cp_1, \ldots, cp_n\}$ be the exported names of L.

Let $names(L) \doteq imp(L) \cup exp(L)$ be the names of L.

Let $imports(L) \doteq \Gamma_0$ be the import environment of L.

Let $exports(L) \doteq \phi, cp_1 : \langle \kappa,\ cp_1,\ cp_1$',\ field^1*,\ mdecl^1*,\ bdecl^1* \rangle\ |\ \langle \epsilon, cp_1, field^1$*, $c_1 \rangle$ $, \ldots, cp_n : \langle \kappa, cp_n, cp_n$',\ field^n*,\ mdecl^n*,\ bdecl^n* \rangle\ |\ \langle \epsilon, cp_n, field^n$*, $c_n \rangle$ be the export environment of L.

The predicate $linkset(L)$ corresponds to coherence of the predicate L, i.e. whether the linkset L uses the names coherently such that it does not break linking.

Definition 3.5.2 *(Linksets)*

Consider the structure $L \equiv \Gamma_0\ |\ cp_i \blacktriangleright \Gamma_i \vdash \tau_i^{i \in 1...n}$.

$linkset(L) \Leftrightarrow$

- *$env(imports(L))$, and $env(exports(L))$*

- *$\forall i \in 1 \ldots n$, we have $env(\Gamma_0, \Gamma_i)$ and $dom(\Gamma_i) \subseteq exp(L)$*

- *$imp(L) \cap exp(L) = \phi$.*

where env is a predicate on an environment which holds true when all the names in the environment is unique [13]. Formally,

$$env(\phi, x_1 : t_1, \ldots, x_n : t_n) \Leftrightarrow \forall i, j \in 1 \ldots n, i \neq j \Rightarrow x_i \neq x_j$$

The predicate $intra\text{-}checked(L)$ corresponds to separate compilation of each of the fragments in L. It is valid only if typechecking does not fail with the given environment.

Definition 3.5.3 *(Intra-checked linksets)*

Let $L \equiv \Gamma_0 \mid cp_i \blacktriangleright \Gamma_i \vdash \tau_i^{i \in 1 \ldots n}$.

$intra\text{-}checked(L) \Leftrightarrow$

- $linkset(L)$

- $\Gamma_0 \vdash \diamond$ *and,* $\forall i \in 1 \ldots n$, *we have* $\Gamma_0, \Gamma_i \vdash \tau_i$.

The predicate $inter\text{-}checked(L)$ corresponds to inter-module typechecking and ensures that the fragments can be linked in a type-safe way.

Definition 3.5.4 *(Inter-checked linksets)*

Let $L \equiv \Gamma_0 \mid cp_i \blacktriangleright \Gamma_i \vdash \tau_i^{i \in 1 \ldots n}$.

$inter\text{-}checked(L) \Leftrightarrow$

- $intra\text{-}checked(L)$

- $\forall j, k \in 1 \ldots n$, cp, Γ', Γ'', *if Γ_k has the form Γ', $cp : \langle \kappa, cp, cp', field*, mdecl*, bdecl* \rangle \mid \langle \epsilon, cp, field*, c \rangle$, Γ'' and $cp \equiv cp_j$ then $\langle \kappa, cp, cp', field*, mdecl*, bdecl* \rangle \mid \langle \epsilon, cp, field*, c \rangle \equiv \langle \kappa, cp_j, cp_j', field^j*, mdecl^j*, bdecl^j* \rangle \mid \langle \epsilon, cp_j, field^j*, c_j \rangle$*

Merging two compatible linksets is a useful operation. We will first see the merging operation on environments before we move on to merging of linksets.

Definition 3.5.5 *(Environment compatibility and merge)*

- $\Gamma \backslash X$ *is the environment obtained from Γ by removing the assumptions $cp : \langle \kappa, cp, cp', field*, mdecl*, bdecl* \rangle \mid \langle \epsilon, cp, field*, c \rangle$ such that $cp \in X$.*

- $\Gamma \upharpoonright X$ *is environment obtained from* Γ *by retaining only the assumptions* $cp : \langle \kappa, cp, cp', field^*,$
  $mdecl^*, bdecl^* \rangle \mid \langle \epsilon, cp, field^*, c \rangle$ *such that* $cp \in X$.

- *Compatible environments:* $\Gamma_1 \div \Gamma_2 \leftrightarrow \forall cp \in dom(\Gamma_1) \cap dom(\Gamma_2),$ *we have* $\Gamma_1(cp) = \Gamma_2(cp)$.

- *Merge of two environments is defined as* Γ_1 *and* Γ_2 *as* $\Gamma_1 + \Gamma_2 \doteq \Gamma_1, (\Gamma_2 \backslash dom(\Gamma_1))$.

The following lemma states that merge operation on environments is commutative.

Lemma 3.5.1 *(Commutation of environment merge)*

If $\Gamma_1 \div \Gamma_2$ *and* $\Gamma, (\Gamma_1 + \Gamma_2), \Gamma' \vdash \tau$, *then* $\Gamma, (\Gamma_2 + \Gamma_1), \Gamma' \vdash \tau$.

Proof

$E_1 + E_2$ *is just a permutation of* $E_2 + E_1$ *under the assumption* $E_1 \div E_2$.

The merge of two linksets is defined as follows. The new linkset's global environment will shrink since some of the names of both linksets become internal to the new linkset. Correspondingly, the environments of individual fragments will expand to include the names of the other linkset that got merged.

The following lemmas show some of the properties of a merged linkset. The proofs are very similar to the one given by Cardelli [13].

Definition 3.5.6 *(Linkset merge)*

Let $L \equiv \Gamma_0 \mid cp_i \blacktriangleright \Gamma_i \vdash \tau_i^{i \in 1...l}, L' \equiv \Gamma_0' \mid cp_i' \blacktriangleright \Gamma_i' \vdash \tau_i^{i \in 1...n}$. *If* $linkset(L), linkset(L')$, *and* $exp(L) \cap exp(L') = \phi$, *then:*

$L + L' \doteq \Gamma_0 \backslash exp(L') + \Gamma_0' \backslash exp(L) \mid cp_i \blacktriangleright \Gamma_0 \upharpoonright exp(L'), \Gamma_i \vdash \tau_i^{i \in 1...l}, cp_i' \blacktriangleright \Gamma_0' \upharpoonright exp(L), \Gamma_i' \vdash$
$\tau_i'^{i \in 1...n}$

Lemma 3.5.2 *(Linkset merge)*

If $linkset(L), linkset(L')$, *and* $exp(L) \cap exp(L') = \phi$, *then* $linkset(L + L')$.

Lemma 3.5.3 *(Intra-checked merge)*

If $intra\text{-}checked(L), intra\text{-}checked(L'), imports(L) \div imports(L')$, *and* $exp(L) \cap exp(L') = \phi$,
then $intra\text{-}checked(L + L')$.

Definition 3.5.7 *(Linkset compatibility)*

$L \div L' \Leftrightarrow$

$imports(L) \div imports(L'), imports(L) \div exports(L'), imports(L') \div exports(L), and\ exp(L) \cap$

$exp(L') = \phi.$

Lemma 3.5.4 *(Inter-checked merge)*

If $inter\text{-}checked(L)$, $inter\text{-}checked(L')$, and $L \div L'$, then $inter\text{-}checked(L + L')$.

3.6 Modules as linksets

Module judgements represent modules while linksets represent set of fragments to be linked. Both module judgements and linksets are collection of fragments. So, it is natural to convert module judgements to linksets so that if the converted linksets are inter-checked, we can infer that the module is separately compilable.

Consider the following module judgement for example from Figure 3.1.

```
(Object : ⟨κ, Object,⊤, φ, φ, φ⟩,
Collection : ⟨κ, Collection, Object, _, _, _⟩
Iterator : ⟨κ, Iterator, Object, _, _, _⟩
Integer : ⟨κ, Integer, Object, _, _, _⟩
Node : ⟨κ, Node, Object, _, _, _⟩
List : ⟨κ, List, Collection, Node head, Node y,
List add(Integer), Iterator iter(), φ⟩,
ItemAdded : ⟨ε, ItemAdded, List list, Integer item, List ⟩)
⊢
(List : ⟨κ, List, Collection, Node head, Node y,
List add(Integer), Iterator iter(), φ⟩ ≐ FElement♯)
∴
(List : ⟨κ, List, Collection, Node head, Node y,
List add(Integer), Iterator iter(), φ⟩)
```

This can be converted to the following linkset where the environment of the module judgement becomes the interface of the linkset.

```
Object : ⟨κ, Object,⊤, φ, φ, φ⟩,
Collection : ⟨κ, Collection, Object, _, _, _⟩
```

```
Iterator : ⟨κ, Iterator, Object, _, _, _⟩
Integer : ⟨κ, Integer, Object, _, _, _⟩
Node : ⟨κ, Node, Object, _, _, _⟩
List : ⟨κ, List, Collection, Node head, Node y,
List add(Integer), Iterator iter(), φ⟩,
ItemAdded : ⟨ε, ItemAdded, List list, Integer item, List ⟩
| List ▶ List♯ : ⟨κ, List, Collection, Node head, Node y,
List add(Integer), Iterator iter(), φ⟩,\\
```

I formally define the translation as follows.

Definition 3.6.1 *(Compilation of a module)*

$$\lceil \Gamma \vdash M \therefore S \rceil \doteq \Gamma \mid \lceil S \vdash M \therefore S \rceil'$$

$$\lceil \Gamma \vdash \phi \therefore \phi \rceil' \doteq \texttt{empty fragment list}$$

$$\lceil \Gamma \vdash (cp : s \doteq cp\sharp, M) \therefore (cp : s, S) \rceil' \doteq cp \blacktriangleright findEnv(\mathcal{D}(\{cp\}), \Gamma) \vdash cp\sharp : s, \lceil \Gamma \vdash M \therefore S \rceil'$$

The auxiliary function $findEnv$ is defined as follows:

$$findEnv(names, \Gamma) \doteq \{cp : s \mid cp \in names \cap dom(\Gamma) \wedge \Gamma(cp) = s\}$$

The formal definition of dependency function $\mathcal{D}$ (adapted from [3]) is given in Appendix.

Informally it gives, for any set of fragment names $cp*$, the set $cp*'$ of all the fragment names whose type information is required for the compilation of the fragments in $cp*$.

The following lemma establishes the relationship between a module judgement and its corresponding linkset.

Lemma 3.6.1 *(Properties of compilation)*

If $L \equiv \lceil \Gamma \vdash M \therefore S \rceil$, then $imports(L) = \Gamma$ and $exports(L) = S$.

Proof: Clearly, $imports(\lceil \Gamma \vdash M \therefore S \rceil) = imports(\Gamma \mid \lceil S \vdash M \therefore S \rceil') = \Gamma$. If $L \equiv \Gamma \mid U$, $exports(U) \doteq exports(L)$. I prove that if $U \equiv \lceil \Gamma' \vdash M' \therefore S' \rceil'$, then $exports(U) = S'$ by induction on the translation of $\lceil \Gamma' \vdash M' \therefore S' \rceil$.

Case $\lceil \Gamma' \vdash \phi \therefore \phi \rceil'$**.** We have $U \equiv \lceil \Gamma' \vdash \phi \therefore \phi \rceil' \equiv$ empty fragment list. Hence, $exports(U) = exports(\Gamma \mid U) = \phi = S'$.

Case $\lceil \Gamma' \vdash (cp : s \doteq cp\sharp, M'') \therefore (cp : s, S'') \rceil'$**.** We have $U \equiv cp \blacktriangleright findEnv(\mathcal{D}(\{cp\}), \Gamma') \vdash cp\sharp : s, U'$ and $U' \equiv \lceil \Gamma' \vdash M'' \therefore S'' \rceil'$. Hence, $exports(U) = cp : s, S''$.

58

3.7 Properties of Separate Compilation

Cardelli mentions two important properties of separate compilation [13].

- Well-typed modules are compiled to well-typed linksets.

- Two well-typed modules with compatible interfaces can be safely compiled and merged.

I prove each of these separately in this section.

Theorem 3.7.1 *(Separate Compilation)*

If $\Gamma \vdash M \therefore S$, *then* $inter\text{-}checked(\lceil \Gamma \vdash M \therefore S \rceil)$.

Proof:

The translation of $\lceil \Gamma \vdash M \therefore S \rceil$ produces a structure of the shape $L \equiv \Gamma \mid cp_i \blacktriangleright \Gamma_i \vdash \tau_i^{i \in 1...n}$. We have $dom(S) = dom(exports(L)) = exp(L)$, and $dom(\Gamma) = dom(imports(L)) = imp(L)$. We have that $imports(L) = \Gamma$, and since $\Gamma \vdash M \therefore S$, we have $\Gamma \vdash \diamond$ and $env(\Gamma)$. Therefore

$$\Gamma \vdash \diamond \tag{3.1}$$

$$env(imports(L)) \tag{3.2}$$

Since $exports(L) = S$ and $\Gamma \vdash M \therefore S$, we have $env(S)$. Therefore

$$env(exports(L)) \tag{3.3}$$

By induction on the derivation of $\Gamma \vdash M \therefore S$, for all $i \in 1...n$, we have $\Gamma, \Gamma_i \vdash \tau_i$, and hence $env(\Gamma, \Gamma_i)$. Hence

$$\Gamma, \Gamma_i \vdash \tau_i \tag{3.4}$$

$$env(\Gamma, \Gamma_i) \tag{3.5}$$

By construction, each Γ_i is a subset of S, hence $dom(\Gamma_i) \subseteq dom(S) = exp(L)$; i.e.

$$dom(\Gamma_i) \subseteq exp(L) \tag{3.6}$$

By construction, $dom(\Gamma) \cap dom(S) = \phi$; i.e.

$$imp(L) \cap exp(L) = \phi \tag{3.7}$$

Statements (3.2), (3.3), (3.5), (3.6) and (3.7) prove that

$$linkset(L) \tag{3.8}$$

Statements (3.8) and (3.4) prove that

$$intra\text{-}checked(L) \tag{3.9}$$

Note that throughout the construction of $\lceil - \rceil'$, the list of signatures S is maintained and a subset of that will be the local environment of any newly created linkset fragment. This along with the statement (3.9) prove that

$$inter\text{-}checked(L) \tag{3.10}$$

Since $inter\text{-}checked(\lceil \Gamma \vdash M \therefore S \rceil)$ produces $L \equiv \Gamma \mid cp_i \blacktriangleright \Gamma_i \vdash \tau_i^{i \in 1...n}$, we have $inter\text{-}checked(\lceil \Gamma \vdash M \therefore S \rceil)$.

This theorem proves that well typed modules in Ptolemy (classes and event types) can be transformed into corresponding well-typed linksets. Since in Cardelli's framework, modules are compared to source module language and linksets to target language of the compilation, this essentially proves that separate compilation holds in Ptolemy. In fact, it would be possible to model code generation similar to that in [4] by adding code generation rules along with typing rules.

To prove the next theorem which is that two separately compiled modules with compatible interfaces can be safely compiled and merged, I define the compatibility of signature and module judgements.

Definition 3.7.1 *(Signature and module compatibility)*

$(\Gamma \vdash S) \div (\Gamma' \vdash S') \doteq \Gamma \div \Gamma', \Gamma \div S', \Gamma' \div S,$ and $dom(S) \cap dom(S') = \phi$

$(\Gamma \vdash S) \div (\Gamma' \vdash d' \therefore S') \doteq (\Gamma \vdash S) \div (\Gamma' \vdash S')$

$(\Gamma \vdash M \therefore S) \div (\Gamma' \vdash S') \doteq (\Gamma \vdash S) \div (\Gamma' \vdash S')$

$(\Gamma \vdash M \therefore S) \div (\Gamma' \vdash M' \therefore S') \doteq (\Gamma \vdash S) \div (\Gamma' \vdash S')$

Lemma 3.7.1 *(Compatibility under compilation)*

Assume $(\Gamma \vdash M \therefore S) \div (\Gamma' \vdash M' \therefore S')$. Then, $\lceil \Gamma \vdash M \therefore S \rceil \div \lceil \Gamma' \vdash M' \therefore S' \rceil$.

Proof: By definition 3.7.1, $(\Gamma \vdash M \therefore S) \div (\Gamma' \vdash M' \therefore S') \doteq (\Gamma \vdash S) \div (\Gamma' \vdash S')$, and hence $\Gamma \div \Gamma'$, $\Gamma \div S'$, $\Gamma' \div S$, and $dom(S) \cap dom(S') = \phi$. Take $L \equiv \lceil \Gamma \vdash M \therefore S \rceil$ and $L' \equiv \lceil \Gamma' \vdash M' \therefore S' \rceil$. By Lemma 3.6.1, $imports(L) \div imports(L')$, $imports(L) \div exports(L')$, $exports(L) \div imports(L')$, and $exp(L) \cap exp(L') = \phi$. Therefore, by Definition 3.5.7, $L \div L'$.

I now show that compatibility of signatures is a sufficient condition for the safe merge of separately compiled modules.

Theorem 3.7.2 *(Separate compilation and merge)*

Assume $\Gamma \vdash M \therefore S$, $\Gamma' \vdash M' \therefore S'$, *and* $(\Gamma \vdash S) \div (\Gamma' \vdash S')$. *Then, inter-checked*$(\lceil \Gamma \vdash M \therefore S \rceil + \lceil \Gamma' \vdash M' \therefore S' \rceil)$.

Proof: Let $L \equiv \lceil \Gamma \vdash M \therefore S \rceil$ and $L' \equiv \lceil \Gamma' \vdash M' \therefore S' \rceil$. By Theorom 3.7.1, we have *inter-checked*(L) and *inter-checked*(L'). Since $(\Gamma \vdash S) \div (\Gamma' \vdash S')$, we also have $(\Gamma \vdash M \therefore S) \div (\Gamma' \vdash M' \therefore S')$ by Definition 3.7.1. By Lemma 3.7.1 we obtain $L \div L'$. Therefore, by Lemma 3.5.4, we have *inter-checked*$(L + L')$.

This theorem proves that two compatible modules compiled separately can be linked together safely. A useful scenario is when one module consists of subject classes and the other consists of observer classes. Since they depend on each other through the event types, they can be modified without disturbing the other module as long as the event types remain the same. However if the event types change, classes in both these modules will have to change to accommodate it just like if any class changes its interface, all depending classes have to adapt.

CHAPTER 4.   RELATED WORK

In this chapter, we look at the other works that are related to my work. I divide the chapter into two sections to describe related work for each of my two parts of the work.

4.1   Case Study

Several others have evaluated different characteristics of AO approaches. For example, early assessments of modularity benefits were conducted [35, 27, 51]. Mendhekar et al. [35] used RG, an environment for creating image processing systems to evaluate aspect-oriented programming. Kersten and Murphy [27] used Atlas, a web-based learning environment to evaluate aspect-oriented programming.

Walker et al. [51] also conducted an initial assessment of aspect-oriented programming where they show that AspectJ may improve the understandability when the effect of the aspect code has well-defined scope and that AOP could change the strategies used by programmers to address tasks associated with aspect code. Hannemann and Kiczales [24] compared the object-oriented and aspect-oriented implementations of the Gang of Four design patterns [22] using qualitative metrics. Garcia et al. [1] used a set of quantitative metrics to compare the object-oriented and aspect-oriented implementations.

The closest related work is by Lesiecki [32] that makes observations about the incremental compilation time of AspectJ programs. Compared to these earlier results, the empirical study presented in this work rigorously analyzes the AO test-driven software development process that serves to evaluate the potential utility of load-time deployment.

4.2 Separate Compilation in Ptolemy

The closest related work is the seminal paper on separate compilation by Luca Cardelli [13]. There he formalizes the notion of separate compilation. He simplifies separate compilation to separate type-checking which is modeled by a judgement $\Gamma \vdash a : A$ which means that a program fragment a is of type A where the environment Γ contains the type information of the related fragments of a. In his framework, he maps judgements of the form $E \vdash a : A$ to what he calls linksets which is a simple configuration language to list the collection of fragments to be linked. To describe slightly more structured modules, he introduces bindings, signatures and their corresponding judgements. I stick to his framework, but adapt it to our work to deal with a more structured modules consisting of classes and event types.

Another very closely related work is by Ancona and Zucca [5] where they show that existence of principal typings [53] guarantee separate compilation for Java-like languages. They define an abstract framework for modeling separate compilation and inter-checking for Java-like languages. They define soundness and completeness of inter-checking which can be checked when the type system has principal typings. Their definition of linkset inter-checking allows selective recompilation which means that after a change in the code, only those program fragments within a given linkset that needs to be compiled will be compiled. Since selective recompilation depends on whether the compiled binary code changes, they include code generation as well in their framework for separate compilation. I did not need their framework for the lesser work of just proving separate compilation for Ptolemy. But whether Ptolemy has principal typings and whether it has the similar advantage in selective recompilation can be a future work. They also require a special type system and hence a generic module system cannot support it.

Ancona et al. have proposed a new type system to support true separate compilation in Java [4] unlike the partial separate compilation supported by the current standard Java compilers. They model both compilation of a specific module using its local type environment and global compilation of the corresponding closed program and prove that the binaries generated for the module remain the same in both. My work does not assume a Java-like implementation of Ptolemy and does not get into code generation.

CHAPTER 5.   FUTURE WORK

There are some possible future works that can be pursued based on the case study and on support for programmer productivity in Ptolemy.

5.1   Case study

5.1.1   IDE support for Test Driven Development

Going forward it would be interesting to see if automatic prediction of the best-suited deployment technique is possible for a given project at the project, module and unit test level. This prediction could be incorporated as a built-in feature of a standard IDE. Such utility could estimate the sum of both compilation time and execution time for both deployment techniques and use the smaller of the two. Our results give some insight into such prediction but many issues remain to be explored.

Section 2.2.8 provide some insight about how we may be able to find the normalized execution time for each technique. There I argued that the normalized execution time is $m * J + n * C * s$, where the constants m and n are dependent on non-differentiating parameters like percentage of join point covered by the aspect code, join point shadow density etc. Therefore, even though the difference in normalized execution time which is same as the difference in execution time is $n * C * s$, we do not know how to compute n. But this research path does not eliminate the inherent disadvantages that both static and load-time weaving have.

5.1.2   Other paths

One of the ways load-time weaving could be improved is to explore if method-level loading granularity can be achieved in virtual machines. In that case, instead of weaving the classes that get executed, only the methods that get executed could be woven thereby further increasing the advantage of weav-

ing at load-time. Another area to explore is to further improve the incremental compilation of AspectJ like languages. But research in fundamental language design to eliminate the causes of sub-optimal incremental compilation will be much better. Ptolemy [41] is a good example for research in that area.

5.2   Programmer productivity in Ptolemy

In Section 4.2, I discussed about the work of Ancona and Zucca [5] related to principle typings in Java-like languages. It would be interesting to see if Ptolemy also has principal typings and hence supports selective recompilation.

CHAPTER 6. CONCLUSION

With the aim of studying the support for programmer productivity in general and test-driven development in particular by some of the programming languages supporting separation of conceptual concerns, I made two specific contributions.

I have presented an empirical assessment studying the adequacy of load-time weaving in decreasing the time spent during edit-compile-test iterations. I showed that in a test-driven development process, neither load-time weaving nor static weaving consistently outperforms the other in every case. The relative performance of these techniques depends on several parameters, namely the size of the project, the number of classes loaded during test execution, and the average join point shadow usage during test execution. In particular, we saw that:

- as the size of the project increases, load-time deployment saves more time during compilation compared to static deployment,

- as the number of classes loaded increases, the test execution time for load-time deployment increases faster than that of static deployment,

- as the average join point shadow usage increases, the test execution time for load-time deployment approaches that of static deployment, and

- the percentage of join point shadows advised by aspects does not differentiate between the two deployment models.

Although it is not possible to say precisely when one technique will start outperforming the other, informally we can say that load-time weaving is likely to perform better for larger projects with mostly small unit tests and static weaving is likely to be better for smaller projects with large unit tests. Such

decisions can be made at three different levels - project level, module level and unit test level, depending on the organization of the project. In summary, our results confirmed what was believed informally that incremental compilation can be an important issue for large aspect-oriented projects. However, contrary to the general perception that load-time weaving solves the problem, I showed that load-time deployment solves this problem only under some favorable conditions.

I also briefly explained the features of Ptolemy and proved the property of separate compilation which is an important property from the point of view of programmer productivity, especially from the point of view of test-driven development.

APPENDIX

Definition .0.1 *(Dependency Function)*

The dependency function $\mathcal{D}$ is defined as follows:

$$\mathcal{D}(CP) = \{cp' \mid \exists cp \in CP \, s.t. \, cp \rightarrow cp'\}$$

where the relation $\rightarrow$ is defined by $cp \rightarrow cp'$ iff $cp' \in refClasses(cp\sharp)$.

$refClasses(\textbf{class }c\textbf{ extends }c'\{field^*\,meth^*\,binding^*\}) = \{c,c'\} \cup refClasses(field^*) \cup refClasses(meth^*) \cup$
$refClasses(binding^*)$

$refClasses(c\textbf{ evtype }p\{field^*\}) = \{c\} \cup refClasses(field^*)$

$refClasses(field_1 \ldots field_n) = \bigcup_{i \in 1 \ldots n} refClasses(field_i)$

$refClasses(c\,f) = \{c\}$

$refClasses(meth_1 \ldots meth_n) = \bigcup_{i \in 1 \ldots n} refClasses(meth_i)$

$refClasses(t_0\,m(t_1\,x_1, \ldots, t_n\,x_n)\{e;\,\} = \{t_0, \ldots, t_n\} \cup refClasses(e)$

$refClasses(binding_1 \ldots binding_n) = \bigcup_{i \in 1 \ldots n} refClasses(binding_i)$

$refClasses(\textbf{when }p\textbf{ do }m) = refClasses(p)$

$refClasses(e_1; e_2) = refClasses(e_1) \cup refClasses(e_2)$

$refClasses(new\,c()) = \{c\}$

$refClasses(e_0.m(e_1, \ldots, e_n)) = \bigcup_{i \in 0 \ldots n} refClasses(e_i)$

$refClasses(t\,var = e_1; e_2) = \{t\} \cup refClasses(e_1) \cup refClasses(e_2)$

$refClasses(e.f) = refClasses(e)$

$refClasses(e.f = e') = refClasses(e) \cup refClasses(e')$

$refClasses(\textbf{event }p\{e\}) = \{p\} \cup refClasses(e)$

$refClasses(\textbf{register}(e)) = refClasses(e)$

$refClasses(\textbf{invoke}(e)) = refClasses(e)$

Figure A.1   Definition of $refClasses$

BIBLIOGRAPHY

[1] Alessandro Garcia and Cláudio SantŠAnna and Eduardo Figueiredo and Uirá Kulesza and Carlos Lucena and Arndt von Staa. Modularizing design patterns with aspects: a quantitative study. In *4th International Conference on Aspect-Oriented Software Development, pp 3–14*, 2005.

[2] G. Ammons. Grexmk: Speeding up scripted builds. In *Fourth Workshop on Dynamic Analysis (WODA)*, 2006.

[3] D. Ancona, G. Lagorio, and E. Zucca. A Formal Framework for Java Separate Compilation. In *ECOOP '02: 16th European Conference on Object-Oriented Programming*, pages 609–636, 2002.

[4] D. Ancona, G. Lagorio, and E. Zucca. True separate compilation of Java classes. In *PPDP '02: 4th international conference on Principles and practice of declarative programming*, pages 189–200, 2002.

[5] D. Ancona and E. Zucca. Principal typings for Java-like languages. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 306–317, New York, NY, USA, 2004. ACM.

[6] B. Atherton et al. The Ant project: a Java based build tool. `http://ant.apache.org/`.

[7] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. abc: an extensible aspectj compiler. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 87–98, New York, NY, USA, 2005. ACM Press.

[8] K. Beck. *Test-driven Development: By Example*. 2003.

[9] C. Bockisch, M. Arnold, T. Dinkelaker, and M. Mezini. Adapting virtual machine techniques for seamless aspect support. *SIGPLAN Not.*, 41(10):109–124, 2006.

[10] C. Bockisch, M. Haupt, M. Mezini, and R. Mitschke. Envelope based weaving for faster aspect compilers. In *Net.ObjectDays*, 2005.

[11] J. Bonér. What are the key issues for commercial AOP use: how does AspectWerkz address them? In *3rd International Conference on Aspect-Oriented Software Development*, pages 5–6, 2004.

[12] G. Bracha and W. Cook. Mixin-based inheritance. In *OOPSLA/ECOOP '90*, pages 303–311, 1990.

[13] L. Cardelli. Program fragments, linking, and modularization. In *POPL '97: 24th Principles of Programming Languages*, pages 266–277, 1997.

[14] C. Clifton. A design discipline and language features for modular reasoning in aspect-oriented programs. Technical Report 05-15, Iowa State University, Jul 2005.

[15] C. Clifton and G. T. Leavens. MiniMAO$_1$: Investigating the semantics of proceed. *Sci. Comput. Programming*, 63(3):321–374, 2006.

[16] C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein. MultiJava: modular open classes and symmetric multiple dispatch for Java. In *OOPSLA '00: 15th Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 130–145, 2000.

[17] E. W. Dijkstra. On the role of scientific thought. *EWD 477*, August 1974.

[18] R. Dyer and H. Rajan. Nu: a dynamic aspect-oriented intermediate language model and virtual machine for flexible runtime adaptation. In *7th International Conference on Aspect-Oriented Software Development (AOSD)*, 2008.

[19] T. Elrad, R. E. Filman, and A. Bader. Aspect-oriented programming: Introduction. *Commun. ACM*, 44(10):29–32, 2001.

[20] M. Flatt and M. Felleisen. Units: cool modules for hot languages. In *PLDI '98: 1998 conference on Programming language design and implementation*, pages 236–248, 1998.

[21] M. Flatt, S. Krishnamurthi, and M. Felleisen. A programmer's reduction semantics for classes and mixins. In *Formal Syntax and Semantics of Java*, chapter 7, pages 241–269. 1999.

[22] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. 1995.

[23] D. Garlan and D. Notkin. Formalizing design spaces: Implicit invocation mechanisms. In *VDM '91: 4th International Symposium of VDM Europe on Formal Software Development-Volume I*, pages 31–44, 1991.

[24] J. Hannemann and G. Kiczales. Design pattern implementation in Java and AspectJ. In *17th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 161–173, 2002.

[25] M. Haupt, M. Mezini, C. Bockisch, T. Dinkelaker, M. Eichberg, and M. Krebs. An execution layer for aspect-oriented programming languages. In *VEE '05: 1st ACM/USENIX international conference on Virtual execution environments*, pages 142–152, 2005.

[26] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *OOPSLA '99*, pages 132–146.

[27] M. Kersten and G. C. Murphy. Atlas: a case study in building a web-based learning environment using aspect-oriented programming. In *OOPSLA '99: 14th Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 340–352, 1999.

[28] Kiczales and Erik Hilsdale and Jim Hugunin and Mik Kersten and Jeffrey Palm and William G. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *ECOOP 2001*, pages 327–353. June 2001.

[29] Kiczales and John Lamping and Anurag Mendhekar and Chris Maeda and Cristina Lopes and Jean-Marc Loingtier and John Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming (ECOOP)*, June 1997.

[30] G. Kniesel, P. Costanza, and M. Austermann. Jmangler-a framework for load-time transformation of Java class files. In *SCAM 2001*, pages 100–110, 2001.

[31] B. B. Kristensen and K. Osterbye. Roles: conceptual abstraction theory and practical language issues. *Theor. Pract. Object Syst.*, 2(3):143–160, 1996.

[32] N. Lesiecki. Applying AspectJ to J2EE application development. In *AOSD '05*, 2005.

[33] D. C. Luckham, J. J. Kennedy, L. M. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and analysis of system architecture using Rapide. *IEEE Transactions on Software Engineering*, 21(4):336–54, April 1995.

[34] E. M. Maximilien and L. Williams. Assessing test-driven development at IBM. In *ICSE '03: 25th International Conference on Software Engineering*, pages 564–569, 2003.

[35] A. Mendhekar, G. Kiczales, and J. Lamping. RG: A case-study for aspect-oriented programming. Technical Report SPL97-009 P9710044, Xerox PARC, February 1997.

[36] D. Notkin, D. Garlan, W. G. Griswold, and K. J. Sullivan. Adding implicit invocation to languages: Three approaches. In *JSSST International Symposium on Object Technologies for Advanced Software*, pages 489–510, 1993.

[37] H. Ossher and P. Tarr. Multi-dimensional separation of concerns using hyperspaces. IBM Research Report 21452, IBM, April 1999.

[38] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–8, December 1972.

[39] A. Popovici, T. Gross, and G. Alonso. Dynamic weaving for aspect-oriented programming. In *AOSD '02*, pages 141–147, 2002.

[40] H. Rajan. Design patterns in Eos. In *PLoP '07, Conference on Pattern Languages of Programs*, September 2007.

[41] H. Rajan and G. T. Leavens. Ptolemy: A language with quantified, typed events. In *ECOOP '08: 22nd European Conference on Object-Oriented Programming*, July 2008.

[42] H. Rajan and K. J. Sullivan. Eos: instance-level aspects for integrated system design. In *ESEC/FSE-11*, pages 297–306, Sept 2003.

[43] H. Rajan and K. J. Sullivan. Classpects: unifying aspect- and object-oriented language design. In *ICSE '05*, pages 59–68, 2005.

[44] K. Sakurai, H. Masuhara, N. Ubayashi, S. Matsuura, and S. Komiya. Association aspects. In *AOSD '04*, pages 16–25, 2004.

[45] N. Scharli, S. Ducasse, O. Niersstrasz, and A. P. Black. *Traits: Composable Units of Behaviour*, volume 2743/2003. Springer Berlin / Heidelberg, 2003. Lecture Notes in Computer Science.

[46] R. B. Setty, R. Dyer, and H. Rajan. Weave now or later: A test-driven development perspective on aspect-oriented deployment models. *Submission to ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2008.

[47] M. Stoerzer and J. Graf. Using pointcut delta analysis to support evolution of aspect-oriented software. In *ICSM '05: 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 653–656, 2005.

[48] M. Surtani, J. Greene, and B. Ban. The JBoss Cache project: open source data grids. `http://labs.jboss.com/jbosscache/`.

[49] D. Suvée, W. Vanderperren, and V. Jonckers. Jasco: an aspect-oriented approach tailored for component based software development. In *2nd International Conference on Aspect-Oriented Software Development pp. 21–29*, 2003.

[50] T. Tourwé, J. Brichau, and K. Gybels. On the existence of the aosd-evolution paradox. In L. Bergmans, J. Brichau, P. Tarr, and E. Ernst, editors, *SPLAT: Software engineering Properties of Languages for Aspect Technologies*, March 2003.

[51] R. J. Walker, E. L. A. Baniassad, and G. C. Murphy. An initial assessment of aspect-oriented programming. In *ICSE '99: 21st international conference on Software engineering*, pages 120–130, 1999.

[52] M. Wand, G. Kiczales, and C. Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. *ACM Trans. Program. Lang. Syst.*, 26(5):890–910, 2004.

[53] J. B. Wells. The essence of principal typings. In *ICALP '02: Proceedings of the 29th International Colloquium on Automata, Languages and Programming*, pages 913–925, London, UK, 2002. Springer-Verlag.

[54] E. Wohlstadter, S. Jackson, and P. Devanbu. Dado: Enhancing middleware to support crosscutting features in distributed, heterogeneous systems. *ICSE '03*, 00:174, 2003.

[55] T. Xie and J. Zhao. A framework and tool supports for generating test inputs of aspectj programs. In *AOSD '06: 5th International Conference on Aspect-Oriented Software Development*, pages 190–201, 2006.

[56] T. Xie, J. Zhao, D. Marinov, and D. Notkin. Detecting Redundant Unit Tests for AspectJ Programs. In *17th International Software Reliability Engineering (ISSRE'06)*, pages 179–190, 2006.

[57] G. Xu. A regression tests selection technique for aspect-oriented programs. In *WTAOP '06: 2nd workshop on Testing aspect-oriented programs*, pages 15–20, 2006.

[58] G. Xu and A. Rountev. Regression test selection for AspectJ software. In *ICSE '07: 29th International Conference on Software Engineering*, pages 65–74, 2007.

[59] J. Zhao, T. Xie, and N. Li. Towards regression test selection for AspectJ programs. In *WTAOP '06: 2nd workshop on Testing aspect-oriented programs*, pages 21–26, 2006.