On Aspect-oriented Programming for Enforcing Software Design Rules

A dissertation presented
by

Pengcheng Wu

to the Faculty of the Graduate School
of the College of Computer and Information Science
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

Northeastern University
Boston, Massachusetts

May, 2010

Abstract

Software design rules are important in modern software development, with significance in achieving high quality in many aspects of software engineering including functional correctness, safety, performance, reusability, and so on. The current practice of software engineering tools to enforce software design rules has much to be desired. They usually either can only check a pre-defined set of design rules without allowing customization and extension, or even when customization or extension is allowed, they require programmers to encode new design rules in low level primitives such as byte-code level API, or force programmers to learn new domain specific languages or new meta-models that programmers are not familiar with.

In this thesis work, we argue that the shadow model of AspectJ-like Aspect-oriented programming languages is a useful and suitable meta-model for building customizable software design rule static checkers for programs written in Java-like object-oriented languages. To support this thesis, based on AspectJ's shadow model and pointcut language, we have built two compile time facilities, with which software design rules can be encoded and enforced. The first facility is called Statically Executable Advice, which allows programmers to implement design rule checkers in advice-like constructs, but unlike advice in AspectJ, they are advice that are defined on static shadows, instead of on dynamic join points, and that they are executed at compile time, instead of at run time. The second facility is a fur-

ther improvement of the idea, by featuring a Datalog based declarative query mechanism on shadows, with a seamless integration with AspectJ's native pointcut expression language. When designing this Datalog shadow query mechanism, we especially developed it with a focus on performance and scalability. We present a Datalog representation of the shadow model of AspectJ so that we can leverage an intelligent data structure capable of dealing with large-scale relational data with much redundancy, called Binary Decision Diagram (BDD). We make use of an advanced third party BDD-based Datalog solver called "bddbddb" to solve design rule constraints encoded in Datalog shadow queries in our system. We have evaluated our Datalog-based approach for enforcing software design rules in terms of its effectiveness, usability, and performance. The evaluation results show that the system can describe and enforce a wide variety of industrial software design rules, it is easier to use than writing queries on an alternative meta-model, i.e., the Abstract-Syntax-Tree based model, and the system indeed can scale well to large size real world programs on diversified queries.

Acknowledgement

First of all, I would like to thank my thesis advisor, Prof. Karl Lieberherr, for his direction and encouragement over the years, including the years when I was at Northeastern University and the years when I worked in industry while trying to reach this ultimate academic goal. It is fair to say that without his support, I would not have been able to finish this PhD work. I am also grateful to the rest of my thesis committee members, including Professors Gene Cooperman, Shriram Krishnamurthi, and Mitchell Wand, for their time and effort spent on reading drafts of my dissertation, giving feedback and providing helpful suggestions. In particular, I want to thank Prof. Wand for his detailed comments and many useful suggestions on the Evaluation chapter of the dissertation.

During my Northeastern years, I am also indebted to many people, from whom I have learnt much. I had a collaboration with Prof. David Lorenz and the collaboration has resulted in my first published academic paper during Northeastern period, the AOSD'03 paper co-authored with Karl and David. The main contents of the paper have turned into Chapter 2 of this dissertation, and David has made significant contribution to it. In the summer of 2001 and summer of 2004, I also had an honor to work with Dr. Dirk Riehle and Dr. Rajesh Krishnan respectively as an intern in industry. I have tremendously benefited from the working experience with them and I thank them for treating me well. I am also grateful to Neeraj Sangal for allowing

me to work with him in the summer of 2003. Neeraj's entrepreneurship spirit has always inspired me since then.

I am indebted to the contributors of the Eclipse AspectJ compiler, the BDD-based Datalog solver bddbddb, and the Eclipse plug-in ASTView for making their software and source code available. Difference pieces of the software tools presented in this dissertation are built on their software.

Over the years, I have made many friends at Northeastern, in the Greater Boston area, and in China. They are too many to list here. Chatting and playing with these friends have made life enjoyable and colorful.

My family is always the most important aspect of my life. Words cannot really express my gratitude to my wife, Xin, for her love, understanding, and encouragement along this journey. Even though still a young kid, Ethan, my lovely son, has made contribution to this work by being cooperative, cute and funny. This is for you, Xin and Ethan! I thank my parents for their love and their appreciation for education, which made this possible in the first place. I thank my sister for her support and encouragement. My parents-in-law certainly deserve my gratitude for believing that I can finish this PhD work. It is such a pity that my mother-in-law was not able to see this happen eventually.

Contents

List of Figures

List of Tables

Listings

CHAPTER 1

Introduction

Modern software systems rely on conformance to various type specifications, software design rules, architectural standards, and semantic contracts for the whole system to be functional, robust, reliable, and comprehensible. Automating checking of their conformance is of importance to software quality.

While many software errors, in particular, non-conformance with regard to type specifications, can be detected by modern programming language compilers, a lot more are not being checked by compilers. Just as an example, Java's [20] `java.lang.Object` class has two important methods that can be overridden by derived classes, i.e., `hashCode` and `equals`, and to ensure those derived classes can work properly with the hash table based collection classes, there are important programming guidelines that the programmer must follow, and that are statically checkable. In his well known book, *Effective Java* [8], Joshua Bloch synthesizes two Java programming guidelines that state one must *always override hashCode when one overrides equals* and when they are both overridden, *they should better use the same set of fields*. Following the two guidelines will help ensure the program to obey the `Object` class' important contract, which states that when two objects are equal according to the `equals` method, then calling `hashCode` on the two objects must return the same integer [57].

Failure to obey this contract has serious consequences,e.g., one will not be able to retrieve back the object that she has just put into a hash table, which is of course almost always undesirable. Despite being such an important programming guideline for Java programmers, yet we are not aware of any compiler that checks programs' conformance to it.

This kind of programming guidelines sometimes are also called *Software Design Rules*.While there is no formal definition of the term, informally it can be characterized as [49]: *Software design rules constrain the structure or behavior of a program and express desirable programming practices [49].* This thesis introduces a new way to build static checkers to enforce Java program's conformance to a set of design rules.

1.1 Examples of design rules

There are various reasons to impose design rule constraints on programs. We give some examples of software design rules grouped by categories below. They are for facilitating discussions and motivating our work, and so the list of the categories is not exhaustive.

- Design rules to ensure functional correctness
  - Java classes' `equals` methods must be defined on a parameter of type `Object`. Failure to obey this rule will result in the class' `equals` method not being called when interacting with hash based collection classes, and thus the program will behave incorrectly.
  - A Java class providing a `clone` method definition should implement the `Cloneable` interface. Failure to do so will get a `CloneNotSupportedException` exception at run time.

- Design rules to ensure program safety
  - Programs intended to be deployed to embedded systems should avoid recursive call chains. Such an example exists in the MISRA guidelines adopted by the automobile industry [3]. The rule is intended to avoid programs running out of stack space when the call chain gets too deep.
  - Programs intended to be deployed as real-time embedded systems should avoid using dynamic memory allocation. An example of this rule can also be found in the MISRA guidelines. The intent of the rule is to avoid running out of memory at run time and the nondeterministic latency of dynamic memory allocation that is unacceptable for real time systems.
- Design rules to help boost program performance
  - In Java, one should avoid using string concatenations inside a loop. In such a case, use the `StringBuffer` class instead [8]. The reason for this rule is that Java strings are immutable and thus when two strings are concatenated, the two strings are actually copied into a newly created dummy string object, which can be very inefficient if executed frequently.
- Design rules to help boost reusability of methods
  - Use an abstract type for a method parameter, if all of the parameter access inside the method implementation can be done through the abstract type [16]. This way, more types of objects can be passed in as the parameter to the method.

- Design rules to ensure architectural conformance
  - The Law of Demeter [41] is such an example for the purpose of decreasing coupling between modules.

1.2 Current Design Rule Checking Practices and Deficiencies

Because software design rules usually are not checked by compilers, users have to turn to specialized static checkers if they want to enforce design rules on their programs. The current practices of design rule checkers have much to be desired.

The first approach of achieving design rule checking is to use tools like Lint [33]. Lint was a tool originally developed for checking legal but suspicious constructs in C language source code, but later the same idea has been applied and extended to other languages like Java. Lint-like tools provide a rich set of questionable code patterns that can be checked, but they lack the capability of letting users customize and extend the rule set. Lacking this capability is a critical deficiency, given no one can possibly foresee how many design rules are out there or how many more will be coming out.

The second approach is exemplified by tools like FxCop [16] and FindBugs [53]. FxCop is Microsoft's .NET framework code assembly static analyzing tool. It checks and reports information about a code assembly, such as possible design, localization, performance, and security issues [16]. FindBugs [27, 53] is a popular bug finding tool among Java programmers. Many bug patterns that it can detect are related to design rule violations. Both tools provide rich sets of design rules or bug patterns that they can detect, and at the same time they also allow users to add their customized ones by using the .NET

introspection APIs or Visitor pattern style APIs plus the Byte Code Engineering Library (BCEL) [2] respectively. It has been reported [49] that this kind of extension approach is very difficult to use due to the complexity of the APIs. For example, Listing 1.1 is an example [22] of a Visitor method used by the FindBugs tool to identify all `isLogging` method calls made on the `Logger` class in code. The reason that those method calls are of interest is that logging calls are expensive to execute and thus they should be constrained from being used.

Listing 1.1: A code example in FindBugs

```
public void sawOpcode(int seen) {
   if(‘‘cbg/app/Logger’’.equals(classConstant) &&
      seen == INVOKESTATIC && ‘‘isLogging’’.equals(nameConstant) &&
      ‘‘()Z’’.equals(sigConstant)) {
      seenGuardClauseAt = PC;
      return;
   }
}
```

When writing this code, the programmer first needs to understand how the whole Visitor framework works. In particular, the programmer needs to know that `sawOpcode` is a method that can be overridden and that will be called by FindBugs during a visit to the method body of a method implementation, and the byte code symbol of the operation visited will be passed in as the argument. Second, the programmer has to grasp the byte code manipulation APIs to understand what a `class constant` is, what a `name constant` is, what a `signature constant` is and what the instruction code is for each instruction concerned, e.g., `INVOKESTATIC` is the instruction code symbol for a static method call. It is clear that a more declarative and higher level abstraction is needed and will be helpful.

As the third approach, users could turn to a general purpose code

query system to write queries against the underlying model and express the intended design rule or bug patterns. Examples of such kind of code query systems include ASTLog [17], Java Tools Language (JTL) [15], JQuery [31], and CodeQuest [23]. The problem with this approach is that there is no unified underlying data model that those query systems can operate on, and thus users will have to learn each individual newly created data model and/or a new query language.

1.3 Our approach

Our approach fits into the third approach category as discussed above, but instead of building a new data model about the program, we reuse AspectJ-like Aspect-oriented Programming (AOP) [36, 35] languages' shadow model as the underlying data model, and extend its pointcut expression query mechanism to enable expressive software design rule checking.

Our thesis statement is that **AspectJ-like Aspect-oriented Programming (AOP) languages' shadow model is a useful and suitable framework to build user customizable software design rule static checkers.** To support this thesis statement, we have designed and implemented two linguistic query extensions based on AspectJ's shadow model. The first extension allows Visitor pattern style compile time advice to be defined on selected program shadows, while the second extension features a Datalog based pointcut query system operating on a shadow database extracted from the underlying program, with an intelligent data structure called Binary Decision Diagrams representing the queried shadows.

While more evidence need to be lay out to support the thesis statement in the rest of the dissertation, the high level reason for

this determination is that we believe there is an intrinsic connection between AOP's shadow model and what is needed for specifying constraints of software design rules. In particular, as a technique aimed at improving modularity of software implementations with cross module nature, AOP languages have provided linguistic mechanisms to allow programmers to query or talk about programming elements across modules and their relationships, which lend itself to being a good basis of design rule specifications. Our earlier work [34] suggests that AspectJ's dynamic join point model, which is the run time representation of the static shadow model, is able to capture the essence of a design rule, i.e., the Law of Demeter. The benefits of using the shadow model for a design rule checker implementation include:

- It is a useful and rich abstraction of object-oriented program structures, with less important syntactical details omitted;
- It can be ported to multiple base languages, as the shadow model has been increasingly adopted by Aspect-oriented variations of C++ [58], C# [54], and MATLAB [61];
- It is familiar to average AOP programmers.

To support the thesis statement, we have designed and implemented two linguistic query extensions to allow users to detect design rule violations. The first extension, called *Static Executable Advice* [34, 66], allows programmers to make use of the exposed compile time shadow information to write customized design rule static checkers in Java, running against program elements selected by using the regular AspectJ pointcut designator expressions. The user supplied static checkers are attached to the AspectJ compiler to run during the aspect weaving process.

More recently, we have designed and implemented a second extension, which we find more useful and usable. It features a Datalog-based query language, operating on the extensional database (EDB) of program shadow records, collected during the compilation process of the AspectJ compiler. User supplied Datalog programs can be written to capture static shadows satisfying the negation of a desired design rule constraint, which indicates that a violation of the design rule has been found. As a strength of the logic programming paradigm, the Datalog-based approach delivers declarativeness and succinctness that are very desirable but are missing in imperative approaches.

In addition, when designing and implementing the Datalog-based extension, one of our primary design considerations is that the approach and the implemented system should scale well to large real world programs. We observed there exists redundant information among program shadow records that can be leveraged by a new advanced Binary Decision Diagram (BDD) [11] based Datalog solver *bddbddb* [65, 39], in the interest of boost the system's scalability. We have specially designed the Datalog extensional database schema such that its structures are suitable for being represented using the intelligent BDD data structure, and for the Datalog program to be efficiently solved by *bddbddb*.

To evaluate the effectiveness, we apply the approach on many practical software design rules to show customized static checkers can be implemented. Our usability study shows that the approach is superior to the alternative approach, i.e., writing Datalog queries on the Abstract Syntax Tree based data model. Our performance study also shows that the approach can indeed scale to large sizes of real world applications.

1.4 Thesis Contributions

This dissertation makes the following contributions to the Aspect-oriented programming area and to the software engineering area:

1. We identified AspectJ's shadow model as a useful and suitable underlying data model to build extensible software design rule checkers.

2. We designed and implemented a system in the frame work of the AspectJ language and its industrial strength compiler, the Eclipse AspectJ compiler, to allow users to capture various design rule violations.

3. We identified that the BDD is a suitable representation of the shadow model so that we can leverage an advanced BDD based Datalog solver *bddbddb* to make the system scale to large size of applications.

4. Together with others' work [5], we proposed and argued for using Datalog as the query language on shadow model to achieve declarative queries.

5. We conducted evaluations to show that our proposed approach can be used to effectively implement real world design rule detection algorithms, that the approach is more usable than querying on the alternative data model, i.e., the Abstract Syntax Tree model, and that the system indeed can deal with large size real world programs.

1.5 Outline

The organization of the rest of this dissertation is as follows. Chapter 2 presents two AspectJ-based dynamic checkers for two variants of

the Law of Demeter, serving as motivational examples to show why AOP languages are appropriate basis for developing program design rule checkers. Chapter 3 briefly introduces the first proposed extension to the AspectJ language, Statically Executable Advice, and shows how it can be used to solve the motivational example, but still has much to be desired. Chapter 4 introduces the design and implementation of our second proposed extension to AspectJ, the Datalog-based shadow query system, and shows how it can be used to implement practical design rule checkers. Chapter 5 discusses the evaluations that we carried out to validate our approach and system. Chapter 6 discusses the related work. Chapter 7 concludes the dissertation and discusses possible future work.

CHAPTER 2

Dynamic Checkers for the Law of Demeter

As examples to motivate our work, we present how we can implement dynamic checkers for two forms of one software design rule, the Law of Demeter, in AspectJ. In particular, for one form that in theory is statically checkable, we show that although the AspectJ's pointcut designator language is not expressive enough to support the query that is necessary for implementing a static checker for it, the information needed to implement such a checker is already present in its shadow model. This finding suggests AspectJ's shadow model may serve as a good basis for building static checkers for software design rules. Most of the material in this chapter has already been presented in our published paper [34].

2.1 The Definition of the Law of Demeter

The Law of Demeter (LoD) is a design rule proposed to decrease coupling between program components [42]. In this design rule, call sites between OO components constitutes coupling between them. The LoD states which couplings are acceptable and which are best avoided. Informally, LoD states that an object or a class should only

talk to "closely related" objects or classes respectively, thus leading to less coupled OO systems [28]. There have been two forms of the LoD suggested, i.e., the object form (OF) and the class form (CF).

OF states that an object can only invoke method calls on: itself, the arguments of the enclosing method context, its instance variables, a locally constructed object within the enclosing method context, or a returned object from a method call made to itself.

CF, on the other hand, states that in the implementation of a class's method, one should only call the class's other methods or methods of the classes of its arguments, instance variables, classes used to locally instantiate instances, and the classes that are return types of methods in the class. OF is intended to be more restrictive than CF in the sense that OF cares about particular objects while CF only cares about types.

In general, validating OF must be done dynamically, and validating CF can be done dynamically, or statically (by analyzing the source code).

2.2 Validating LoD in AspectJ

We are first interested in using aspect-oriented programming (AOP) techniques, especially AspectJ, to dynamically check a program's conformance to LoD. There are three good reasons why one would want to check LoD using AOP:

1. Detecting LoD violations is a cross-cutting concern, which is the primary application area for AOP, as it involves checking all method calls in a program. A non-AOP implementation of such a dynamic checker would require invasive modifications to the source code so that one can insert dynamic checking code around each callsite.

2. LoD is easy to express in the join point model of AspectJ, as will be shown later.

3. Checking LoD violations is an interesting, non-trivial application of AOP technology helping to drive it further.

Second, we would like to experiment with the AspectJ language to see how close or how far away it is for us to develop a static checker for the class form of LoD using its `declare error` mechanism, given this form is statically checkable in theory. Ideally, we wish we were able to express the detection of violations of the class form of LoD, in AspectJ, as a pointcut named `LoDViolation`, and make use of the AspectJ language's `declare error` mechanism to report all the violations in the program when being compiled with the following aspect in Listing 2.1. For those readers that are not familiar with AspectJ's `declare error` statement, it reports a compile time error if any program element (shadow) being compiled matches with the specified pointcut designator (PCD) expression. Since this is a compile time detection, the specified PCD expression has to be statically determinable, i.e., none of the pointcuts involving run time entities can be used, and that includes *cflow*, *cflowbelow*, *if*, *args*, *this* and *target* pointcuts.

Listing 2.1: LoD violation detection aspect

```
aspect CheckLoD {
  pointcut CFLoDViolation(): SomeStaticPCDExpression;

  declare error: CFLoDViolation():
               "Class form LoD violation detected!";
}
```

Not surprisingly, this attempt is futile in the current implementation of AspectJ. The logic required to detect LoD violations cannot be

associated with a simple static PCD expression supported by the language, not even for the LoD form which is statically checkable. This checking requires a capability more than just statically determining if a static PCD expression refers to the empty set. We would like to investigate what is missing and what is present in the current AspectJ language mechanism with regard to implementing a static checker along this line.

2.3 Dynamic Checker for OF in AspectJ

When programming in AspectJ, programmers generally reason about the problem by thinking about what the "advisable" join points are, so that advice can be applied to the join points selected by the pointcut expressions.

Listing 2.2 is a utility abstract class that defines all the pointcut expressions needed in this implementation (some of them are needed later in the class form checker). Those pointcut expressions pervasively touch programs and make extensive use of property-based pointcut designators. The `scope()` pointcut prevents the aspects from advising the LoD checker code itself, which is generally desired to avoid circular advice. The `SelfCall` pointcut captures the method calls sent to `this` in a method. Other pointcuts are self-explanatory.

2.3.1 Implementation of the OF checker

The actual implementation of the OF checker uses three concrete aspects with one or two short advice each and a few auxiliary methods. The design of the implementation is clean and easy to understand due to the use of AspectJ's dynamic join point model. Figure 2.1 shows the UML diagram of the object form checker.

Listing 2.2: Util.java

```
package lawOfDemeter;
public abstract class Util {
  public pointcut scope(): !within(lawOfDemeter..*)
    && !cflow(withincode(* lawOfDemeter..*(..)));
  public pointcut StaticInitialization(): scope()
    && staticinitialization(*);
  public pointcut MethodCallSite(): scope()
    && call(* *(..));
  public pointcut ConstructorCall(): scope()
    && call(*.new (..));
  public pointcut MethodExecution(): scope()
    && execution(* *(..));
  public pointcut MethodCall(Object thiz,
    Object target): MethodCallSite()
    && this(thiz)
    && target(target);
  public pointcut SelfCall(Object thiz,
    Object target): MethodCall(thiz,target)
    && if(thiz == target);
  public pointcut Set(Object value): scope()
    && set(* *.*) && args(value);
  public pointcut Initialization(): scope()
    && initialization(*.new(..));
}
```

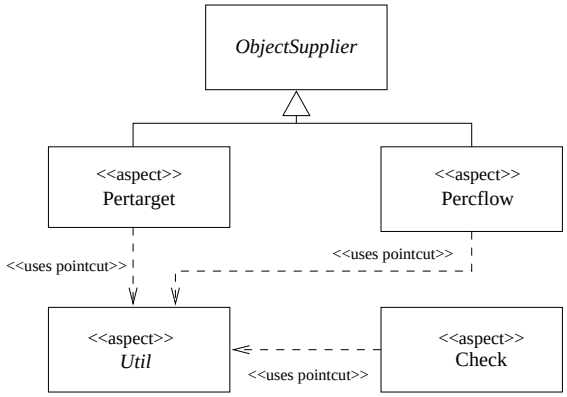

Figure 2.1: The UML diagram of the object form checker

There are two tasks that need to be performed by the checker. One is to collect all of the preferred supplier objects on which methods can be called from an object/context. The other is to verify that each method call makes valid calls on a preferred supplier object of the corresponding “this” object or the context. There are two categories of preferred supplier objects for an object. The first category

Listing 2.3: ObjectSupplier.java

```
abstract class ObjectSupplier {
  protected boolean containsValue(Object supplier){
    return targets.containsValue(supplier);
  }
  protected void add(Object key,Object value){
    targets.put(key,value);
  }
  protected void addValue(Object supplier) {
    add(supplier,supplier);
  }
  protected void addAll(Object[] suppliers) {
    for(int i=0; i< suppliers.length; i++)
      addValue(suppliers[i]);
  }
  private IdentityHashMap targets =
    new IdentityHashMap();
}
```

is context-insensitive: in a method execution on an object, it is legal to call a method on any instance variable of that object. The second category is context-sensitive in that some objects are only preferred in the scope of a method execution, for example, the method call on an argument object is only legal within the method body of the enclosing method.

The class `ObjectSupplier` in Listing 2.3 captures the notion of preferred supplier objects by defining a repository and a set of supporting methods for looking up and adding preferred supplier objects to an object, so that its two sub-aspects can access them.

The aspect `Pertarget` in Listing 2.4 implements the only context-insensitive preferred object situation, i.e., instance variables of an object, by advising the `set` join points. It is declared as `pertarget (Util.Initialization())` so that once a new object o is initialized, an aspect instance of `Pertarget` will be created and associated with o automatically, and each aspect instance can correctly maintain the direct part relationship between the instance variables and their hosting object o. The before-advice on the `set` join points handles with

this logic, in which the `fieldIdentity` method is used so that if an object o_1 has been set as a direct part of an object o_2 through a field f, and later o_2's f is set to another object o_3, we can replace o_1 with o_3 and always maintain the correct direct part relationships for the hosting object o_2.

Listing 2.4: Pertarget.java

```
public aspect Pertarget
  extends ObjectSupplier
  pertarget(Util.Initialization()) {
  before(Object value): Util.Set(value) {
    add(fieldIdentity(thisJoinPointStaticPart),
      value);
  }
  public boolean contains(Object target) {
      return containsValue(target);
  }
  private String fieldIdentity(JoinPoint.StaticPart
    sp) {
    String fieldName = sp.getSignature().
      getDeclaringType().getName() + ":" +
      sp.getSignature().getName();
    if(fieldNames.containsKey(fieldName))
      fieldName=(String)fieldNames.get(fieldName);
    else
      fieldNames.put(fieldName,fieldName);
    return fieldName;
  }
  private static HashMap fieldNames =
    new HashMap();
}
```

The aspect `Percflow` in Listing 2.5, on the other hand, implements all the context-sensitive preferred object situations, by advising `Util .MethodExecution()` to add `this` object and argument objects to the

context sensitive preferred suppliers, and examining results of `Util.SelfCall(Object,Object)` or `Util.ConstructorCall()` to collect the corresponding preferred supplier objects. `Percflow` is intentionally declared as `percflow(Util.MethodExecution())` to simulate the execution scope of a method, instead of requiring manual stack operations.

Listing 2.5: Percflow.java

```
aspect Percflow extends ObjectSupplier
  percflow(Util.MethodExecution()){
  before(): Util.MethodExecution() {
      addValue(thisJoinPoint.getThis());
      addAll(thisJoinPoint.getArgs());
  }
  after() returning (Object result):
    Util.SelfCall(Object,Object)
     || Util.ConstructorCall() {
     addValue(result);
  }
}
```

Finally, the actual checking logic happens in the `Check` aspect in Listing 2.6, which defines the after-advice on method call join points and checks whether the target is a preferred supplier according to LoD.

Listing 2.6: Check.java

```
aspect Check {
  private pointcut IgnoreCalls():
    call(* java..*.*(..));
  private pointcut IgnoreTargets():
    get(static * java..*.*);
  after() returning(Object o):IgnoreTargets() {
    ignoredTargets.put(o,o);
  }
  after(Object thiz,Object target):
    Util.MethodCall(thiz,target)
    && !IgnoreCalls() {
    if (!ignoredTargets.containsKey(target) &&
      !Pertarget.aspectOf(thiz).contains(target) &&
      !Percflow.aspectOf().containsValue(target))
       System.out.println(
         " !! LoD Object Violation !! "
         + thisJoinPointStaticPart);
  }
  private IdentityHashMap
   ignoredTargets = new IdentityHashMap();
}
```

Any design rule has exceptions, including LoD. To make the checker be practically useful, the method calls on some specific objects should be allowed in any situation, e.g., *System.out.println(. . .)* should be allowed to be called anywhere. The `IgnoreTargets` pointcut defines this logic by capturing all those kinds of objects, whose domain currently includes all the public static variables declared in the classes in the packages beginning with `java`. We don't want to check method calls on some stable types either, so we use pointcut `IgnoreCalls` to list those method calls. The `Check` aspect uses the two pointcuts to ignore checking in those two situations. Users can always change those domains by customizing the pointcuts.

From this experiment, we can conclude that a dynamic checker for the object form of the LoD can be elegantly implemented in AspectJ, mostly due to the natural mapping between the dynamic join point model of AspectJ and the definition of the design rule in question. Implementing the same dynamic checker in a non-AOP approach will certainly require significantly more effort.

2.4 Dynamic Checker for CF in AspectJ

As we mentioned earlier, the class form of the LoD is statically checkable in theory, but we could not implement such a static checker by simply using the `declare error` mechanism of the AspectJ, even though this mechanism has been designed to help catch questionable program elements (shadows) in the code. To gain some insights about what is missing and what is present in the current AspectJ language mechanism with regard to implementing a static class form LoD checker, we have implemented a dynamic checker of it in AspectJ.

2.4.1 Implementation of the dynamic CF checker

The class form dynamic checker has a similar functional architecture as the object form checker in that both of them use suppliers and a checker that acts as the client of the suppliers. But from the design point of view, our class form checker uses a different AOP design from the object form checker's. In the object form checker, for each sub-rule, we have corresponding advice. In the class form checker, we have used abstract aspects to specify that when some interesting scenario, which is left unspecified, occurs, some advice will be executed. The concrete sub-aspects reuse the advice defined in the super-aspect by concretizing the interesting scenario. Of course, the concrete sub-

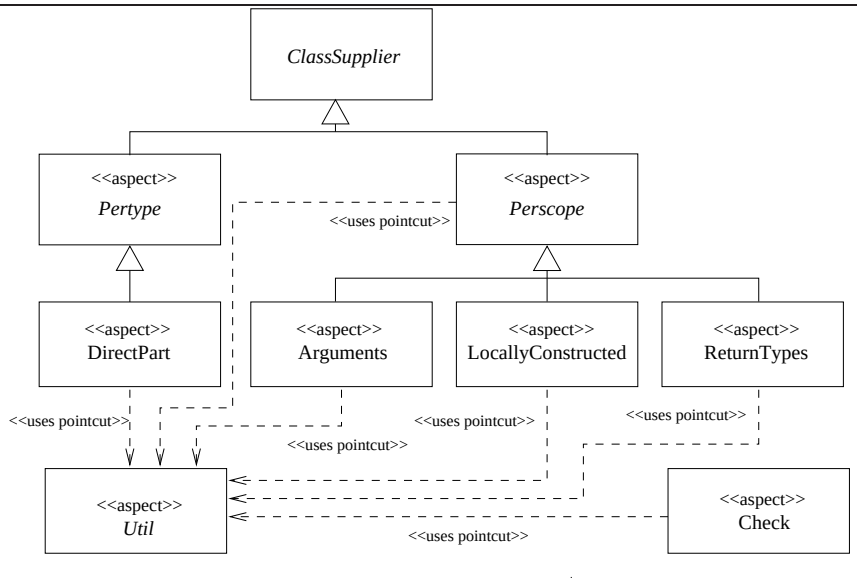

Figure 2.2: *The UML diagram of the class form checker*

aspects can customize the process logic for their scenarios by overriding abstract methods. Figure 2.2 shows the UML diagram of the class form checker.

Listing 2.7: ClassSupplier.java

```
abstract class ClassSupplier {
  protected abstract List
  getSuppliers(JoinPoint.StaticPart enclosingjsp,
    JoinPoint.StaticPart jsp);
}
```

Listing 2.8: Pertype.java

```
abstract aspect Pertype extends ClassSupplier {
  abstract pointcut Pertype();
  before(): Pertype() {
    targets.put(thisJoinPointStaticPart.
      getSignature().getDeclaringType(),
      getSuppliers(thisEnclosingJoinPointStaticPart,
        thisJoinPointStaticPart));
  }
  protected static boolean contains(Class thisType,
    Class targetType) {
    if(targets.containsKey(thisType)) {
      List alloweds = (List)targets.get(thisType);
      Iterator it=alloweds.iterator();
      while(it.hasNext()) {
        if(targetType==it.next())
          return true;
      }
    }
    return false;
  }
  private static HashMap targets = new HashMap();
}
```

Listing 2.9: Perscope.java

```
abstract aspect Perscope extends ClassSupplier {
  abstract pointcut Perscope();
  before() : Util.MethodExecution() {
    st.push(new HashSet());
  }
  before() : Perscope() {
    HashSet aSet = (HashSet) st.peek();
    aSet.addAll(getSuppliers(
      thisEnclosingJoinPointStaticPart,
      thisJoinPointStaticPart));
  }
  after(): Util.MethodExecution() {
    st.pop();
  }
  static boolean contains(Class targetType) {
    HashSet innermost = (HashSet)Perscope.st.peek();
    return innermost.contains(targetType);
  }
  private static Stack st = new Stack();
}
```

The classes and aspects: `ClassSupplier` in Listing 2.7, `Pertype` in Listing 2.8, `Perscope` in Listing 2.9, and `Check` in Listing 2.10 make up an aspect-oriented framework which defines the generic collecting and checking behavior.

We have implemented `Pertype` and `Perscope` as abstract aspects, each of which defines an abstract pointcut (with the same name as the aspect) which is used to collect preferred supplier types. Similar to the situations in the object form checker, the two abstract aspects correspond to the two different situations in which the types are preferred. Table 2.1 lists the correspondences between the two checkers.

The first situation is the context-insensitive situation as defined by `Pertype`, in which some types are always preferred for a given

aspect	object	class
context-insensitive	Pertarget	Pertype and subaspect
context-sensitive	Percflow	Perscope and subaspects

Table 2.1: The correspondences between object/class form checkers.

type. The only context-insensitive situation is the direct part situation, where the types of the instance variables of a class are always preferred in any methods of the class. The second situation is the context-sensitive situation as defined by `Perscope`, in which the types are only preferred when the call sites are in the stack of a particular method execution. (An example of that situation is the arguments situation, where the types of arguments are only legal for the scope of the method body.) All of the concrete aspects extending any of the abstract aspects are supposed to give:

- a definition of the corresponding abstract pointcut to concretize where the advice defined in the super-aspect should be invoked;
- an implementation of the abstract method `getSuppliers` declared in class `ClassSupplier` to expose the preferred types for its particular scenario.

Listing 2.10: Check.java

```
aspect Check {
  private pointcut IgnoreCalls():
    call(* java..*.*(..));
  after(): Util.MethodCallSite() && !IgnoreCalls() {
    Class targetType = thisJoinPointStaticPart.
          getSignature().getDeclaringType();
    Class thisType =
      thisEnclosingJoinPointStaticPart.
        getSignature().getDeclaringType();
    if(!Pertype.contains(thisType,targetType) &&
       !Perscope.contains(targetType))
      System.out.println(
        " !! LoD Class Violation !! "
        + thisJoinPointStaticPart);
  }
```

There are four concrete aspects extending `Pertype` or `Perscope` aspect, which correspond to the four sub-rules of LoD (the "this" class case is combined into the argument case, treating "this" as a special argument) respectively, where a target type is preferred. The implementations of the four concrete aspects are from Listing 2.11 to Listing 2.14. We also allow exceptions to the class form of the LoD, which is defined and configurable by pointcut `Check.IgnoreCalls()`. The `Check` aspect does the straightforward checking logic.

Listing 2.11: DirectPart.java

```
aspect DirectPart extends Pertype {
  public pointcut Pertype():
    Util.StaticInitialization();
  protected List getSuppliers(JoinPoint.StaticPart
    ejsp,JoinPoint.StaticPart jsp) {
    List suppliers=new ArrayList();
    Class currentClass =
      jsp.getSignature().getDeclaringType();
    Field[] fields =
        currentClass.getDeclaredFields();
    for(int i=0; i<fields.length; i++)
        suppliers.add(fields[i].getType());

    return suppliers;
  }
}
```

Listing 2.12: Arguments.java

```
aspect Arguments extends Perscope {
  pointcut Perscope(): Util.MethodExecution();
  protected List
  getSuppliers(JoinPoint.StaticPart ejsp,
  JoinPoint.StaticPart jsp) {
    Class thisClass =
      jsp.getSignature().getDeclaringType();
    List parameterTypes = new ArrayList();
    parameterTypes.add(thisClass);
    parameterTypes.addAll(
      Arrays.asList(((CodeSignature)jsp.
      getSignature()).getParameterTypes()));
    return parameterTypes;
  }
}
```

Listing 2.13: LocallyConstructed.java

```
aspect LocallyConstructed extends Perscope {
  pointcut Perscope():
    Util.ConstructorCall();
  protected List getSuppliers(JoinPoint.StaticPart
    ejsp,JoinPoint.StaticPart jsp) {
    List supplier = new ArrayList();
    supplier.add(jsp.getSignature().
      getDeclaringType());
    return supplier;
  }
}
```

Listing 2.14: ReturnTypes.java

```
aspect ReturnTypes extends Perscope {
  pointcut Perscope(): Util.MethodCallSite();
  protected List
  getSuppliers(JoinPoint.StaticPart ejsp,
    JoinPoint.StaticPart jsp) {
    List supplier = new ArrayList();
    if(ejsp.getSignature().getDeclaringType()!=
      jsp.getSignature().getDeclaringType())
      return supplier;
     supplier.add(((MethodSignature)jsp.
       getSignature()).getReturnType());
     return supplier;
  }
}
```

This implementation is a dynamic checker for CF. However, in Listings 2.7 through 2.14, we only use static type information of classes or methods, as evident from the code that: (1) all of the `getSuppliers` methods are defined on arguments of type `JoinPoint.StaticPart`, which is an AspectJ reflection interface to access the

static shadow information about a dynamic join point; (2) all of the advice are defined on statically determinable pointcuts.

This finding suggests that the AspectJ language's two major components, i.e., the pointcut designator language to indicate the interesting program elements to be processed, and the collected compile time information about those elements (abstracted as the *shadow model*) already have enough compile time information present for statically checking the class form of the LoD, what is missing is just a more expressive query mechanism to leverage this information. This suggests that the shadow information could form a good basis for implementing static checkers for design rules like the LoD, with some extensions to the language and the compiler. This motivates our dissertation work.

CHAPTER 3

A Lightweight Extension to AspectJ: Statically Executable Advice

In this chapter, we would like to extend the AspectJ language so that the static information about join points can be exposed to some compile time facility that is more expressive than AspectJ's `declare error` statement. The goal is that a programmer can make use of this new facility to implement design rule checking by accessing the exposed shadow information.

In this chapter, we present one such compile time facility called, *Statically Executable Advice*, which is a lightweight extension to the AspectJ language. Most material in this chapter has been presented in our published paper [66].

3.1 Dynamic join point model and shadow model

In AspectJ-like AOP languages, *join point model* [35] is a term referring to language mechanisms to specify in a program execution, what runtime events should be caught, and if they get caught, what actions should be taken to change or enhance the behavior of the

program with regard to those events and the exposed context information (dynamic and static) about those events. In AspectJ-like AOP languages, the join point model generally includes a *pointcut designator language*, which is for specifying what runtime events need to be caught, and a mechanism to define *advice*, which is for specifying what actions need to be taken when an intended runtime event does occur. The term, join point model, has dominantly had dynamic nature, and many extensions have been proposed to enhance the dynamic join point model, e.g., [56, 51, 50, 25]. Less research attention, however, has been given to the lexical counterpart of join points, i.e., the shadow model [46, 26].

For each runtime event (join point) in a program execution, there must be a corresponding lexical program element in the code. For example, for any method call event occurring at runtime, there must be a method call node in the program source code. AspectJ and AOP researchers [46, 26] use *shadow* as an abstraction to refer to the lexical counterpart of a dynamic join point. Shadows contain rich static information about the dynamic join points. As an example, a method call shadow contains the name of the method, the static types of the target object and the arguments, and in which method body (which in turn is another shadow) this call is invoked. In AspectJ, there are nine kinds of shadows in the AspectJ language and its compilers [26]. The most common ones are method (or constructor) execution shadows, method (or constructor) call shadows, and field access shadows.

So far shadows have mainly served as internal implementation constructs for building AOP language compilers and for explaining the aspect weaving process [26, 46]. While the notion of shadow is not directly visible or clearly defined for the end user, AspectJ does allow the programmer to access some of those shadow information at runtime in a rudimentary form.

Listing 3.1 is an example AspectJ program with three classes and a logging aspect. Note that at line 21 (and line 22), there is a reference to `thisJoinPointStaticPart` (and `thisEnclosingJoinPointStaticPart` respectively). At runtime, within an AspectJ advice, the `thisJoinPointStaticPart` implicit variable will be bound to the shadow object corresponding to the current executing join point, and the `thisEnclosingJoinPointStaticPart` implicit variable will be bound to the shadow object corresponding to the enclosing join point, if there is any, of the current executing join point. As their names have suggested, they provide *static* information about those join points, and implementation wise, those information has been collected at the compile time by the weaver.

Listing 3.1: Access shadow information at runtime

```
class Foo {
  int compute(Foo a) {
    return 0;
  }
}
class Bar extends Foo {
  void func() {
  }
}
class Main {
  public static void main() {
    Foo f = new Bar();
    Bar b = new Bar();
    f.compute(b);
  }
}

aspect Logging {
  before():
   call(int Foo.compute(..)) {
     System.out.println(thisJoinPointStaticPart);
     System.out.println(thisEnclosingJoinPointStaticPart);
  }
}
```

Those shadow objects retain static information about join points, and thus they provide a different perspective to view the dynamic join points. For example, from the dynamic perspective, a method call join point corresponding to code at line 14 has the signature of `int` f:Bar`.compute(`b:Bar`)`, which reflects the fact that the target object is of type `Bar`, and the argument is of type `Bar` at runtime.However, from the static perspective as offered by the shadow object at line 21, the same method call join point has the signature of `int` :Foo.

`compute(:Foo)`, which is consistent with the view of the code from the Abstract Syntax Tree perspective.

The current AspectJ shadow notion is mainly for internal compiler representation and for simple runtime queries, and thus it is not suitable for a direct exposure to be accessed by more expressive compile time facilities. So we need to adapt and define this shadow model more clearly.

3.2 Our representation of the shadow model

Figure 3.1 is the UML diagram of our proposed abstraction of static join point shadow model, adapted from the AspectJ compiler's internal representation, but retaining the same amount information.

In this model, a program consists of a list of *shadows*, each of which fits into one of the two categories: *NonEnclosingShadow* and *EnclosingShadow*. An *EnclosingShadow* object may contain (lexically) any number of *NonEnclosingShadow* objects while a *NonEnclosingShadow* object is atomic.

Shadows are further classified according to their kinds. The nine concrete classes [1] correspond to the nine kinds of shadows defined in the AspectJ language. Some of the shadow classes implement a signature interface (indicated as circled lines in the figure) so that programmers can access the needed information through well-defined APIs. Those signature interfaces conform to those that are defined in AspectJ's public reflective API package, *org.aspectj.lang.reflect*, except that now we allow those interfaces to be accessible at compile time,

[1] Abstract class names are in the *italic* font, the circled lines represent interface types and the dashed lines represent implementation relationships between classes and interfaces.

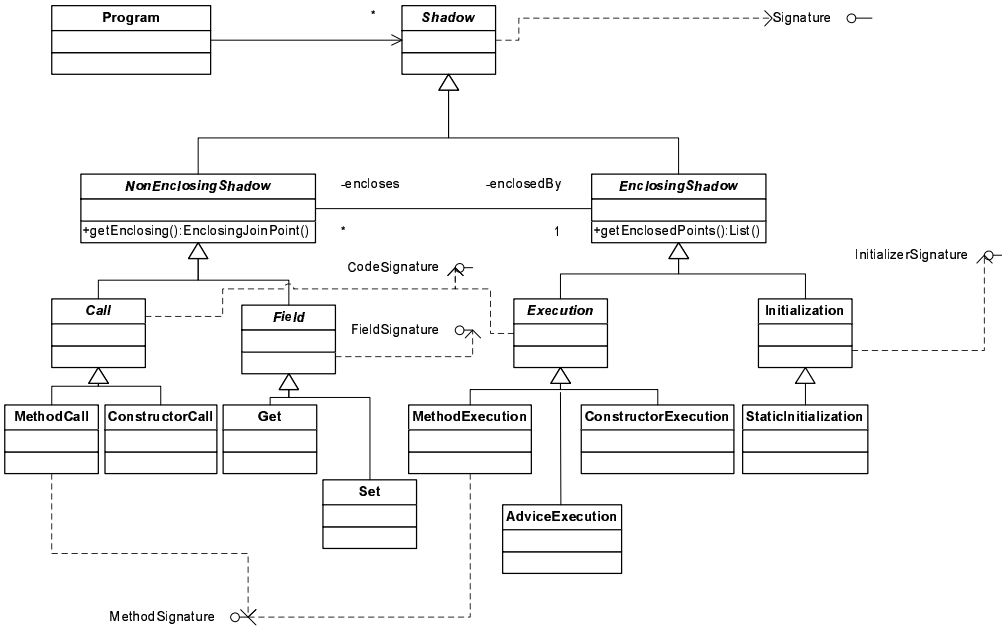

Figure 3.1: Static Shadow Model

instead of just run time in an aspect advice. Those interface APIs should be familiar to average AspectJ programmers. In addition, we have carefully designed the shadow classes so that each shadow class implements the most specialized signature interface applicable. For

example, the `MethodExecution` and `MethodCall` shadow classes implement the `MethodSignature` interface, while the `Field` shadow class implements the `FieldSignature` interface. This delivers a better API usability to programmers, as it avoids many unnecessary type castings that are common in the usage of the current AspectJ language's runtime reflection APIs, where programmers only have direct access to the generic *Signature* interface in advice, and they often have to cast it down to more specialized interfaces after querying about the kind of the current join point.

The signature interfaces lend the programmers the capability to access type hierarchy information as well. The type information comes from Java's reflection system, where class *Class* provides entry points to type information about a loaded class. In the next section, we will see an example how to make use of the type information to reason about a design rule checking problem at compile time.

3.3 Statically Executable Advice

AspectJ's declare error/warning construct is a useful static checking feature, but it is not expressive enough to check complex program properties. Our goal is to make use of AspectJ's *declare* mechanism and to extend it so that more complex user-defined static checking logic can be deployed to execute at compile time. This new proposed language construct is called *Statically Executable Advice*, which can be declared in an aspect definition, and which can access the newly exposed shadow model. A statically executable advice is similar to AspectJ's current runtime advice, except that the former is executed at compile time, and it only has access to statically available information. The following is the syntax of it.

$$\begin{aligned} \textit{AspectDeclStat} &::= \ldots \mid \textit{SEAdv}. \\ \textit{SEAdv} &::= \textbf{declare advice} : \textit{PCD} : \textit{ClassName}. \end{aligned}$$

We add one declare statement kind, `SEAdv`, to the existing allowed declare statements. A statically executable advice declaration,`SEAdv`, is defined with a pointcut expression, `PCD`, and the name of a class. The `PCD` pointcut expression can be any legal AspectJ pointcut expression as long as it is *statically determinable*, which means the pointcut expression can be fully resolved at compile time [59]. The `ClassName` must refer to a Java class that implements a static checking logic, and for it to be able to be used against the exposed shadows, the class must implement a pre-defined interface `IStaticallyExecutableAdvice` as shown in Listing 3.2.

Listing 3.2: Statically executable advice interface

```
interface IStaticallyExecutableAdvice {
  void before(Execution e);
  void after(Execution e);
  void before(Initialization ini);
  void after(Initialization ini);
  void on(Call c);
  void on(FieldOp f);
  void on(ExceptionHandler e);
  void start();
  void finish();
}
```

For each of the two direct subclasses of class `EnclosingShadow`, namely `Execution` and `Initialization`, there are both `before` and `after` methods declared, while for each of the three direct subclasses of class `NonEnclosingShadow` there is only an `on` method. This is intentionally designed to be this way so that we can mimic the lexical scope of shadows. Also note that we do not provide static advice APIs for all of the nine concrete shadow kinds, e.g., `MethodExecution`. This is due to a design decision to free users from having to define many empty dummy methods when implementing the interface, when they only need to deal with few shadow kinds.

The idea is that for each of the `declare advice` statement, the enhanced AspectJ compiler can load the class indicated by `ClassName` and create an instance from it. During the pointcut matching phase, if a shadow matches the specified pointcut expression `PCD`, which is statically determinable, the compiler will create a corresponding shadow object and call the corresponding method on the instantiated instance with the created shadow object as the argument, in a pre-defined order as shown below ($\Rightarrow$ means "implies" and $\hookrightarrow$ means "precedes"):

$$\forall P_1, P_2, P_1 = \textit{enclose}(P_2) \;\Rightarrow\; \textit{before}(P_1) \hookrightarrow \textit{on}(P_2) \wedge \textit{on}(P_2) \hookrightarrow \textit{after}(P_1)$$

Then there are two distinguished `start` and `finish` methods that will be executed at the beginning and the end (respectively) of the whole compilation process.

One of the major advantages of using the statically executable advice construct is that one can make use of AspectJ's pointcut designator selection mechanism to declaratively specify where in the program she wants to apply the checking logic. Our class form Law of Demeter static checker presented below demonstrates this feature.

3.4 A Case Study: Class form LoD Static Checker in Statically Executable Advice

Listing 3.3 is our implementation of a static checker for detecting class form LoD design rule violations in a Java program. Most part of the implementation should be self explanatory, so we just briefly explain some of the most salient features.

The pointcut expression appearing at line 2 and 3 defines the shadows on which the statically executable advice should be applied, and obviously it is statically determinable. This illustrates a strength of the approach, in that the user can declaratively select the shadows to be checked by leveraging the pointcut designator language, e.g., if we only want to check LoD violations inside a particular Java package, `P`, we could just add a conjunction condition, `within(P.*)` to the expression and then recompile the base program with the aspect.

Note that at the end of the `on(Call c)` static advice, when we have determined the method call is not invoked on one of the *known* allowed target types at that point, we could not mark the call to be a violating call immediately, instead, we could only mark it to be a potential violating call. That is because later in the enclosing context method body, there could well be some other constructor types or return types that will be counted as allowed types, which will make the call legitimate according to the LoD. We could only tell if a call is a real violation for sure, when all of the shadows within the enclosing method body have been encountered, and that is why we report violations only in the `after` advice on `Execution`.

Listing 3.3: Static LoD checker in statically executable advice

```
aspect LoDCheckerAspect{
  declare advice : (execution(* *.*(..)) || call(* *.*(..)) ||
                    call(*.new(..))) && withincode(* *.*(..))
                 : LoDChecker;
}

class LoDChecker implements IStaticallyExecutableAdvice {
  HashSet contextAllowedTypes = new HashSet();
  List potentialViolations = new ArrayList();
  Class thisClass;

  public void on(Call c) {
    if(c instanceof ConstructorCall) {
      //A constructor type called in context is an allowed type.
      contextAllowedTypes.add(c.getDeclaringType());
      return;
    }
    if(c instanceof MethodCall) {
      if(c.getDeclaringType() == thisClass) {
        //The return type from a method call on this class is an
        //allowed type in context.
        MethodCall mc = (MethodCall)c;
        contextAllowedTypes.add(mc.getReturnType());
      }
    }
    java.lang.reflect.Field[] fields = thisClass.getDeclaredFields();
    for(int i=0; i<fields.length; i++) {
      if(c.getDeclaringType() == fields[i].getType())
        return;
    }
    if(contextAllowedTypes.contains(c.getDeclaringType()))
      return;
    potentialViolations.add(c);
  }
  public void before(Execution e) {
```

```
36      contextAllowedTypes.clear();
        thisClass = e.getDeclaringType();
38      //This class and argument types are allowed types in context
        contextAllowedTypes.add(thisClass);
40      Class[] paraTypes = e.getParameterTypes();
        for(int i=0; i<paraTypes.length; i++)
42        contextAllowedTypes.add(paraTypes[i]);
    }
44  public void after(Execution e) {
      Iterator it = potentiallyViolations.iterator();
46    while(it.hasNext()) {
        Call c =(Call)it.next();
48      if(!contextAllowedTypes.contains(c.getDeclaringType()))
          System.err.println("An LoD violation at: " +
50                    c.getSourceLocation());
      }
52    potentialViolations.clear();
    }
54
  // other empty methods are emitted.
56}
```

3.4.1 Observations about the checker

In this section, we make some observations about the class form of LoD static checker implemented using the statically executable advice. The observations are both positive and negative. The negative observations motivate a newer and more preferred approach that will be presented in the next chapter.

Using the statically executable advice approach, we could implement a static checker for the class form of LoD using only the statically available shadow information, that was impossible to do by just using AspectJ's `declare error` statement. Here are some positive observa-

tions we can make about the checker implementation.

- The approach allows us to reason about the program structure and the problem at a very high level, i.e., at the shadow level. An alternative approach generally requires low level reasoning, likely at the abstract syntax tree node level, or having to deal with low level APIs such as Java class byte code manipulation APIs.

- The pointcut designator language provides a declarative way to select shadows to be checked.

- The approach is a modest extension to the existing AspectJ language and constructs, with an easy integration with the AspectJ programming model.

On the other hand, the approach has room for improvement due to the following reasons:

- The approach is imperative by nature and a more declarative approach is certainly desirable. This is evident from at least two facts: (1) we had to deal with the subtlety of the visiting sequence of shadows by first marking a method call a potential violation and then confirming if it is real at the right spot of another static advice; (2) by just looking at the implementation, one could hardly connect the pieces of the implementation with the original definition of the LoD.

- The approach assumes some shadow visiting order that may or may not be the case if we are dealing with another implementation of the AspectJ compiler. For example, for the implementation to be correct, it must be the case that a method call shadow is visited by the aspect weaver *after* its enclosing method exe-

cution (definition) shadow is visited. This may not necessarily be true on a different implementation of the compiler.

Realizing those problems, more recently, we have worked out a more declarative approach based on a similar idea, which is presented in the next chapter and is the main result of this dissertation work.

CHAPTER 4

Datalog based Pointcuts for Design Rule Checking

While using the proposed statically executable advice approach presented in the previous chapter we could implement several interesting program design rule checkers, we have found it has much to be desired.

The first problem is that a static checker implementation using that approach tends to be tedious and fragile, due to its imperative nature. For example, by just looking at the design rule checker listed in Listing 3.3, one can hardly recognize that it is to check the conformance of LoD, and even if one can, she will likely have a hard time to figure out which part of the implementation is for which sub-rule (LoD itself has several sub-rules). In addition, the checker implementation is pretty fragile in that it relies on shadows being traversed and processed in some assumed order, e.g., a `MethodExec` shadow has to be processed before a `MethodCall` shadow contained in it can be processed, which is not necessarily true in a different compiler implementation. A more succinct and declarative approach is certainly desirable.

The second problem of the statically executable advice is that allowing programmers to attach Java code to run with the aspect weav-

ing process is dangerous, as the code may potentially do damages on the compilation, e.g., it may even make the compilation fail to terminate.

4.1 A proposal for Datalog based pointcuts

Realizing those problems, we turn to Datalog [63] as a query language operating on the retrieved program shadow records, and propose to let programmers use Datalog to express design rule constraints. We believe Datalog is suitable for this task, because of its declarativeness and its well balance of expressiveness, safety and efficiency.

The basic idea is that during the AspectJ program compilation, all the relevant program type and shadow information, including call graph information, would be collected and stored as Datalog relations (also called Extensional Database, or *EDB*), and programmers can write Datalog-based pointcuts, essentially Datalog inference rules, to express constraints imposed on shadows that will be selected. Then a Datalog solver is used to solve the constraints and return the set of shadows that satisfy the constraints. Our design goal is that Datalog-based pointcuts can be used in conjunction with native AspectJ language pointcut designators using the common pointcut connectors, such as &&, ||, !, and *cflow* etc., with Datalog-based pointcuts focusing on expressing more semantic constraints and the regular native AspectJ pointcuts focusing on expressing simpler syntax constraints.

4.2 Background Information about Datalog

Datalog [63] is a Prolog like logic language, but with more restrictions and far simpler semantics so that it can be evaluated relatively

efficiently. A Datalog program is a set of rules with the following form:

$$P_0 \ :- \ P_1, \ ..., \ P_n \ .$$

Each rule can be divided into two parts, i.e., the rule head ,P_0, and the rule body on the right hand side of $:-$. The rule head, P_0, must be of the form $R(\overrightarrow{x_0})$, and each x in the argument list $\overrightarrow{x_0}$ must be a variable in the corresponding domain. Each $P_i(1 \leq i \leq n)$ in the rule body is called an atom, which can be either of the form $R(\overrightarrow{x_i})$, or $\neg R(\overrightarrow{x_i})$ (negated form). Those R are relations or predicates defined on the domains of their attribute values. Each attribute in the argument list of an atom can be a variable, a constant value of the corresponding attribute domain, or "_", which stands for a value that we do not care. The above rule should be read as: if a substitution of constants for the variables of the rule makes each P_i in the body true, then the head P_0 with this substitution is a true fact.

The rule body is optional. When a rule head P_0 does not appear in any rule with a body, the predicate represents known facts. All of these kinds of predicates are collectively called *extensional database* or *EDB*. On the other hand, predicates that at least appear once as a head in a rule with a body are collectively called *intensional database* or *IDB*.

Datalog is more expressive than relational algebra, since it allows recursion [63]. A Datalog program is usually evaluated in a bottom-up fashion [62] until a fixed-point is reached, i.e., the size of the *IDB* no longer grows. To gain some intuition about the Datalog language, Listing 4.1 is a simple Datalog program that computes the ancestor relationship between two persons, based on the known parent relationship. The `parent` predicate is a part of the EDB, and it represents the parent/child relationship between two persons. The `ancestor` predi-

cate is a part of the IDB, and it computes the transitive closure of the parent relation, which is the ancestor/descendant relation.

Listing 4.1: A simple Datalog program

```
# parent(par:Person,chi:Person)
# ancestor(anc:Person,des:Person)

parent(john,john jr.).
parent(tom,john).
ancestor(x,y) :- parent(x,y).
ancestor(x,y) :- parent(x,z), ancestor(z,y).
```

The evaluation of the Datalog program will result in the following three `ancestor` records, on top of the two existing `parent` records.

```
ancestor(john,john jr.)
ancestor(tom,john)
ancestor(tom,john jr.)
```

While the semantics of the Datalog language is very intuitive, not all Datalog programs have a meaningful semantics. Listing 4.2 is such a Datalog program that does not make sense.

Listing 4.2: An unsafe Datalog program

```
S(x) :- R(y) .
```

This program will make every x in its domain satisfy S, since there is not any positive constraint put on x on the antecedents of the rules. This kind of program is almost certain to be undesirable. Just as another example, Listing 4.3 demonstrates that a Datalog program can end up never converging and thus will run forever.

Listing 4.3: A non-converging Datalog program

```
S(x) :- R(x) .
R(x) :- P(x), !S(x) .
P(john) .
```

One can convince oneself that the program will never terminate by mentally running the program. In iteration 1, we will have `P(john)` and `R(john)` in the result database since `S(john)` is not true. In iteration 2, `S(john)` will be added to the database since `R(john)` has been true since iteration 1. In iteration 3, however, we will have to remove `R(john)` from the database since `S(john)` is now true. Then in iteration 4, we have to remove `S(john)` since `R(john)` is no longer true, and thus we get to the starting point with only `P(john)` being true and the cycle starts again.

These two problems have been well known in the Datalog research community and to overcome the problems, a restricted variant of Datalog language, called *safe and stratifiable Datalog* [13], has been proposed. Specifically, a safe and stratifiable Datalog program is a Datalog program such that:

- Any variable that appears in the head of a rule must also appear in a non-negated predicate in the body.
- A predicate in negated form must not appear in a recursive rule chain.

The first restriction is to avoid the case of every element in an attribute domain satisfying a predicate trivially, while the second restriction is to avoid the case of the program not being able to converge. In this dissertation, we only use this restricted form of Datalog language, and unless we specify otherwise, we always mean the safe and stratifiable Datalog subset when we use the term Datalog. It is not just a coincidence that almost all of recent Datalog-based research work [23, 5, 15] use this restricted form of the Datalog language.

Researchers have been using Datalog as a query language to solve program analysis problems for a long time. Some examples are presented in [63, 55, 7]. More recently, it has been used to implement

a context-sensitive pointer alias analysis algorithm [65], to do program security checking [39], and to define network access control policies [52]. In a closer area relevant to this dissertation work, researchers have used Datalog to build a code query system [23] and to define the static semantics of the AspectJ pointcuts [5]. Datalog is a natural choice for implementing many program analysis tasks, due to its declarative nature and easiness of calculating transitive closures from relations. Even in an apparently less likely area, researchers are using Datalog to achieve declarative network routing in a distributed network environment [43].

When using Datalog for program analysis, one usually starts by retrieving the relevant program information from the source code and storing it as relations or predicates in the EDB. Those relations should abstract out unimportant program syntax details and should be designed in such a way that is convenient for information retrieval and further inference rule writing. Then problem specific Datalog inference rules, which should encode the analysis logic presented as predicates in the IDB, can be written to deduce the wanted results by applying a Datalog solver on those EDB predicates and the inference rules.

The stored EDB relation records and the computed IDB relation records can potentially be huge for practical programs. For example, in our system that will be presented later, to represent the relevant program shadow and type information from a source program with just 20K lines of code, we can get around 100K EDB relation records, and depending on what kind of IDB inference rules we have, the result database size can easily get to hundreds of thousands relation records for such a moderate program. So it is important that we can efficiently store and query those relations. The good news is that there may be a lot of redundant information in program relation

records, which can benefit from using an intelligent data structure, Binary Decision Diagram (BDD).

4.3 Performance Consideration: BDD-based Representation of Datalog relations

While Datalog programs can be solved relatively efficiently in polynomial time [64, 29], it still poses a performance concern when it is applied to shadow relations generated from practical programs, which can easily get to hundreds of thousands of records for even a moderate program. Our observation is that although the relation records generated from practical programs can be huge, there are a lot of commonalities or redundancy among them too. So using an intelligent data structure, Binary Decision Diagram (BDD) [11] as the underlying representation for shadow relations can greatly improve the performance of Datalog solving. There is a BDD based Datalog solver *bddbddb* [65, 39] that takes advantage of this optimization if the underlying records are structured properly.

4.3.1 Background Information about BDD

The Binary Decision Diagrams (BDD) [11] were originally developed by the model checking community, to efficiently store and manipulate Boolean formulas.

To represent a Boolean formula, a BDD is designed to be a directed acyclic graph with one root node and two terminal nodes, each of which represents 0 or 1 respectively. Each of the non-terminal nodes is labeled with a decision variable (corresponding to a Boolean variable in the formula) whose value can be either 0 or 1. From each

of the non-terminal nodes, there are two outgoing edges, i.e., a high edge and a low edge. To determine if an assignment of the Boolean variables makes the Boolean formula true, one just starts with the root node, follows the high edge if the corresponding decision variable is 1 in the assignment, and the low edge if the corresponding decision variable is 0 in the assignment. If the final reached terminal node is the 1 node, then the Boolean formula is true, otherwise it is false.

It is well known [65] that a Boolean formula and thus a BDD can be used to represent a relation by following simple transform steps:

- Give each element in each attribute domain a unique binary number with each bit corresponding to a decision variable. So a domain with N elements will need $\lceil \lg N \rceil$ decision variables.
- Build up a truth table to reflect a record's existence in the relation.
- Convert the truth table to a Boolean formula.

More formally, we want to represent a relation $R(D_1, D_2, ..., D_n)$ defined on domains $D_i (1 \leq i \leq n)$ with each domain's size being $N_i = \mid D_i \mid (1 \leq i \leq n)$. For each domain, we will need $B_i = \lceil \lg N_i \rceil (1 \leq i \leq n)$ bits to represent elements in that domain and each bit is called a decision variable. A Boolean formula f defined on the decision variables is a true representation of relation R if:

$$f(x_1, x_2, ..., x_{(\Sigma_1^n B_i)}) = 1 \quad \textit{iff}$$

$$R(\overrightarrow{x_1 ... x_{B_1}}, \overrightarrow{x_{B_1+1} ... x_{B_1+B_2}}, ..., \overrightarrow{x_{(\Sigma_1^{n-1} B_i+1)} ... x_{(\Sigma_1^n B_i)}}) = T$$

The BDD corresponding to f is a representation of relation records in R. To see if a record belongs to R, we just feed the binary encoding

of the record to the BDD, and trace a path from the root node to one of the terminal nodes, following the high edge if the corresponding decision variable is 1, and the low edge if it is 0. If the reached terminal node is 1, then the record belongs to the relation; otherwise, it does not.

To gain some intuition about how BDD is used to represent a relation, Table 4.1 is a relational table storing mappings between methods and their corresponding parameter type signatures. In the table, there are eight methods, so accordingly, for the method domain, there are $\lg 8 = 3$ decision variables, which are x_1, x_2, and x_3 respectively. For the purpose of discussion, let's say the parameter type signature domain can hold up to four different parameter signatures. So accordingly, there are two decision variables for the domain, which are y_1 and y_2 respectively. In this particular example as presented in the table, the eight methods only have two different parameter signatures, with dominantly most of them having a parameter signature labeled as 10.

Method ID			Param. Sig. ID	
x_1	x_2	x_3	y_1	y_2
0	0	0	1	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	0

Table 4.1: Exemplary relation of methods and their parameter type signatures

Figure 4.1 illustrates how this relation is encoded in the BDD. In the diagram, a solid line represents a high edge and a dashed line represents a low edge.

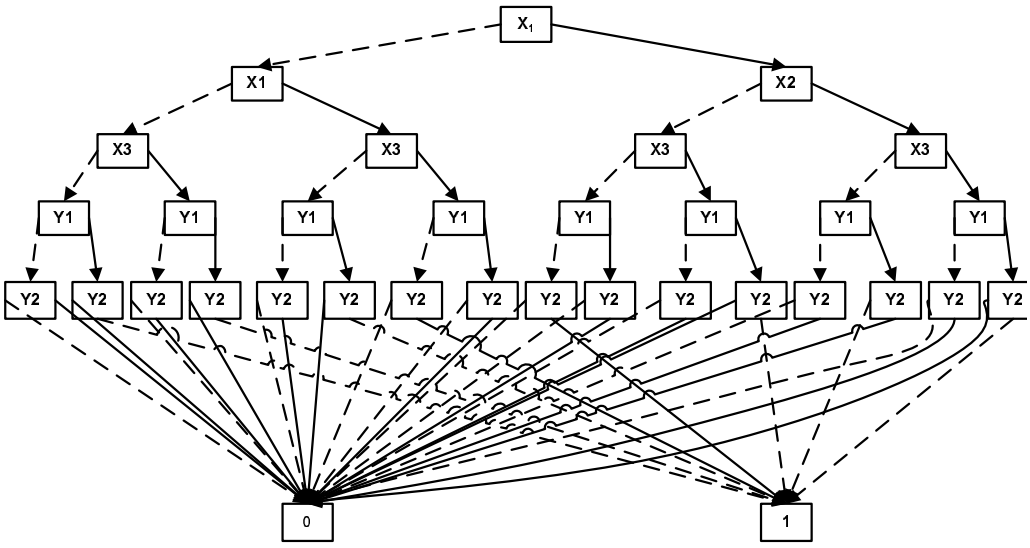

Figure 4.1: The relation encoded in a regular BDD

A BDD itself does not save us any space when used to represent a relation. A particularly useful variant of BDD that can save space is called *reduced ordered binary decision diagrams* (ROBDD) [11], where common BDD subgraphs can be reduced to a single subgraph and the more redundancy there is in the original diagram, the greater memory savings and query improvements the corresponding ROBDD can deliver.

We can get to a ROBDD from a regular BDD by iteratively applying one of the two reductions on the BDD [11] as illustrated in Figure 4.2.

The first reduction says that when two nodes having the same decision variable label have the high and the low edges point to the same node respectively, then the two nodes can be merged into one node that will receive all of the incoming edges to the original two nodes. The second reduction says that when the high edge and the low edge of one node point to the same node, then the former node can be removed from the diagram and all of the incoming edges to it can be directly fed into the latter node.

Figure 4.3 is the result ROBDD after applying the standard reduc-

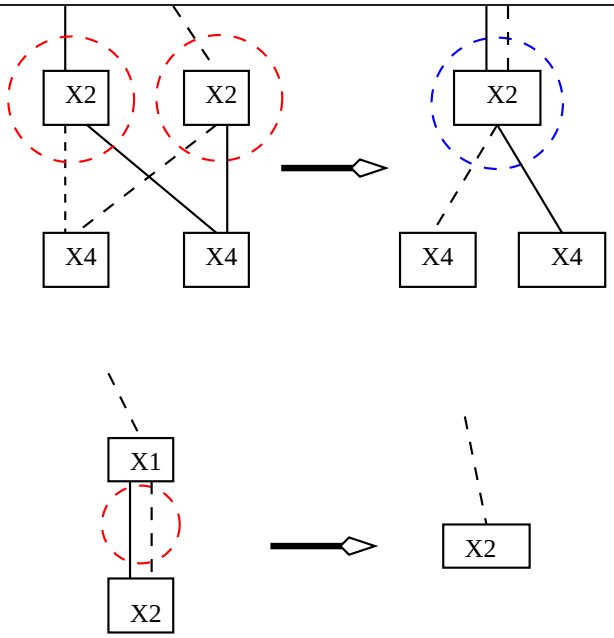

Figure 4.2: Two BDD reduction operations

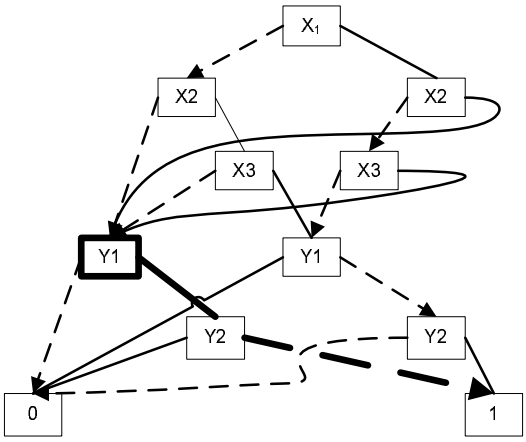

Figure 4.3: The same relation encoded in ROBDD after reductions

tion optimizations on the original BDD in Figure 4.1. Again, a solid line represents a high edge and a dashed line represents a low edge. One can tell how much more compact the ROBDD representation is in this case.

Besides the diagram is much smaller, the other salient feature we observe about this diagram is that all of the paths reaching the terminal node 1 via the $10(Y1Y2)$ sub-path (the edges in boldface) have

joined at the node with the $Y1$ label (the node in boldface), and thus the redundant decision paths have been eliminated. This is corresponding to the fact in the original relation table that most of the methods have the parameter type signature identified as 10, and this characterization has contributed the most to the fact that the result diagram is so compact compared with the original one.

The effectiveness of the reduction steps on a BDD is dependent on two factors. The first factor is how much redundant information there is in the original BDD. The more redundant information is there, the more effective the reductions will be. The second factor is the order of the decision variables, which can make an exponential difference between result ROBDD's. Unfortunately, it has been known that finding the optimal BDD decision variable order is an NP-complete problem [9].

Our goal is to use ROBDD to represent a potentially huge number of Datalog AOP shadow EDB/IDB relation records and to use the advanced BDD based Datalog solver *bddbddb* to solve the inference rules. But the question is: is there much redundant information in the AOP shadow model so that the reductions can be leveraged?

4.3.2 Is There Much Redundancy Among Shadow Records?

The answer is that it depends on how we design the representation of the shadow information, which will be presented a little later. But conceptually, there is abundant redundancy among the program type and shadow relations in which we are interested in the AOP shadow model. For example, to encode program shadow relations, in the EDB that will be generated from source programs, we need to store an important relation that maps a method or a constructor to its cor-

responding parameter type signature, which includes the number of formal parameters, each formal parameter type and its position in the argument list. In any real world program, the number of different method/constructor parameter type signatures could be big, but it is typical that they are not uniformly distributed across methods/constructors. Instead, we will often find that the majority of the methods/constructors take only few variants of parameter type signatures. For example, the parameter type signature of zero parameter, the signature of single parameter whose type is `int`, and the signature of single parameter whose type is `java.lang.String` appear far more frequently than some other signatures. This kind of relation can create many opportunities for the two BDD reductions to kick in, as we have seen in the case of Table 4.1.

To show that this relation is indeed like what we have just described in real world programs, we have collected the method and parameter type signature relationship data from two real world programs available from *sourceforge.net*, which are *jEdit* and *JigSaw*. The former is a Java based edit program, while the latter is a Java based IDE program. Both of them have around 150K lines of code.

The *jEdit* benchmark has 5,016 method definitions with 793 different parameter type signatures, but the top 10% frequent signatures already cover 76% of all methods, and especially the top 5 frequent signatures cover 52.5% of all methods. Table 4.2 lists the top 5 frequent signatures for the *jEdit* benchmark and their occurrences and percentages among the methods.

It is a similar story for the *JigSaw* benchmark. It has 5,781 method definitions with 690 different parameter type signatures, but the top 10% frequent signatures already cover 80% of all methods, and especially the top 5 frequent signatures cover 58.1% of all methods. Table 4.3 lists the top 5 frequent signatures for the *JigSaw* benchmark

Signature pattern	Occurrences	Percentage
(void)	1,855	37%
(String)	351	7%
(int)	228	4.5%
(boolean)	109	2.2%
(ActionEvent)	88	1.8%
	Total	52.5%

Table 4.2: Top 5 frequent method parameter type signatures for jEdit

and their occurrences and percentages among the methods.

Signature pattern	Occurrences	Percentage
(void)	2,385	41.3%
(String)	530	9.2%
(int)	182	3.2%
(Request)	131	2.3%
(Object)	123	2.1%
	Total	58.1%

Table 4.3: Top 5 frequent method parameter type signatures for JigSaw

Another example of this kind of relations is the relation to encode what fields a class has declared, and what their types and type modifiers are. Some field types, e.g., Java primitive types and Java utility collection classes, are much more common than other types. In this relationship, the BDD would have many fields point to the few field types and thus create abundant opportunities for the reductions to kick in. Another similar example is that some methods, e.g., those methods defined in Java collection classes, are invoked more frequently across the program source code.

One more example that can be seen later is that in this work, we also intend to store and infer the program call graph information, where method calls belonging to the same generic function will be statically resolved to a set of method definitions. Method calls with similar receiver types and parameter type signatures will end up with

a similar resolved method definition set, and this kind of commonalities can be leveraged by the BDD reductions as well.

Those kinds of relations are ideal for being stored in the ROBDD form. In reality, there are also enough recent research work [65, 6, 40] suggesting BDD indeed can help improve the scalability of program analysis algorithms.

In addition, native BDD operations can be used to efficiently implement relational algebra operations such as join and projection [65]. This is another reason to use BDD as the underlying representation of the shadow relations.

4.4 The Design of Datalog-based Pointcut System

4.4.1 Shadow model in Datalog

Datalog relations are defined on their attribute domains. We first need to decide which domains that we are going to have and operate on.

4.4.1.1 Shadow Domains

When making decisions about what domains we should have, we followed two design principles. One is that we want to abstract out commonly shared information to be a separate domain, so that we can best leverage the underlying BDD data structure's capability to decrease memory footprint when values of this domain are associated with many records in a relation. The other principle is that we should separate domains in such a way that we can present program shadow relations that are similar to the AspectJ language's join point

model, because it is the way how AspectJ programmers reason about a programming task.

In this system, we have chosen to have the following domains:

`T`: Domain of types, including classes, interfaces, array types and primitive types

`SH`: Domain of program shadows

`Sig`: Domain of program shadow signatures

`PSig`: Domain of method parameter type signatures

`S`: Domain of strings, including string literals and identifiers

`Mod`: Domain of type or construct modifiers

`Z`: Domain of finite integers

It should be obvious why we need most of the domains. The less obvious ones are the domain of `SIG` and the domain of `PSIG`.

The domain of `SIG` contains the collection of records that characterize the relevant static information about shadows, called *signatures*. This domain abstraction is due to two reasons that correspond to our two design principles. The first reason is that different shadows that share the same static signature can just point to the same signature record, e.g., the same method called in different places in the source code can share the same method shadow signature. The second reason is that it has a natural mapping to the concept in the current AspectJ language's reflection API, which allows the programmer to access the static shadow information about the current join point at run time. In that API, the static shadow information is accessed through various *Signature* interfaces depending on the shadow kinds, which in our system will be reflected as a signature predicate can be

further refined into more specific predicates depending on shadow kinds, as will be shown later.

The domain of PSIG contains the collection of records that characterize the static parameter type signature of methods and constructors in the program, and it is specially designed to be a separate domain because we have observed earlier that few parameter type signatures are referred to much more often than the rest of the majority of the signatures. As a comparison, a similar previous work [5] does not have these two abstractions. More difference between the two systems is discussed in section 6.1.

4.4.1.2 Shadow Representation in EDB

Datalog Extensional DataBase (EDB) predicates reflect the known facts that are relevant about a problem domain. In this work, we are interested in presenting the AspectJ's shadow model using a Datalog EDB so that we can write interesting inference rules against this EDB to find out if there is any program element violating a desirable software design rule.

The idea is that the shadow EDB will be collected and emitted from the source program being compiled, during an early stage of the aspect weaving and compilation process. In AspectJ, only shadows occurring in the base program and the input aspects are weavable and thus only those shadows are being collected and analyzed in the compilation and weaving process. But in order to weave advice correctly, an AspectJ compiler also needs to have access to types that are accessed by the base program and the aspects, not just the types that are defined by them. Following this convention, the EDB predicates that we are interested in in this work fit into two categories: the program type information and the shadow information. The former

provides the essential information about types that are defined in the base program or the aspects, or that are used by the base program or the aspects. The latter provides the essential information about the shadows occurring in the base program or the aspects.

We have designed the following type related EDB predicates, whose informal semantics are explained below.

`TypeInfo(t:T, name:S).` `TypeInfo(t,name)` is true iff the program defines or accesses type `t` whose name is `name`.

`TypeModifiers(t:T, mod:Mod).` `TypeModifiers(t,mod)` is true iff type `t` has type modifiers identified as `mod`. Modifier related predicates will be presented later.

`IsClass(t:T).` `IsClass(t)` is true iff type `t` is a Java class.

`IsInterface(t:T).` `IsInterface(t)` is true iff type `t` is a Java interface.

`IsAspect(t:T).` `IsAspect(t)` is true iff type `t` is an AspectJ aspect.

`Package(t:T, pname:S).` `Package(t,pname)` is true iff type `t` is defined in a package whose package name is `pname`.

`IsNested(t:T).` `IsNested(t)` is true iff type `t` is a nested type.

`IsGeneric(t:T).` `IsGeneric(t)` is true iff type `t` is a generic type.

`TypeParameter(t:T,pos:Z,tv:S).` `TypeParameter(t,pos,tv)` is true iff type `t` is a generic type and it has type parameter `tv` in the position of `pos` in its type parameter list.

`IsArray(t:T).` `IsArray(t)` is true iff type `t` is an array type.

`ArrayComponentType(t:T,ct:T).` `ArrayComponentType(t,ct)` is true iff type `t` is an array type and its element type is `ct`.

`DeclaresField(t:T,ft:T,fn:S,mod:Mod).` `DeclaresField(t,ft,fn,mod)` is true iff type `t` declares a field whose static type is `ft`, whose name is `fn` and with modifiers identified as `mod`.

`DeclaresMethod(t:T,rt:T,mn:S,mod:Mod,ps:PSig).` `DeclaresMethod(t,rt,mn,mod,ps)` is true iff type `t` declares a method whose static return type is `rt`, whose name is `mn` with modifiers identified as `mod` and whose parameter type signature is identified as `ps`.

`DeclaresConstructor(t:T,mod:Mod,ps:PSig).` `DeclaresConstructor(t,mod,ps)` is true iff type `t` declares a constructor with modifiers identified as `mod` and whose parameter type signature is identified as `ps`. Note that a constructor does not have a return type and it does not need to have a name, since it is implied by the declaring type name.

`CodeSignatureNumParams(ps:PSig,num:Z).` `CodeSignatureNumParams(ps,num)` is true iff parameter type signature `ps` has `num` parameters.

`CodeSignatureParam(ps:PSig,pos:Z, pt:T).` `CodeSignatureParam(ps,pos,pt)` is true iff parameter type signature `ps` has a parameter with static type of `pt` in the position of `pos` in the parameter list.

`Extends(subT:T,supT:T).` `Extends(subT,supT)` is true iff type `subT` is a class or interface, type `supT` is a class or interface, and type `subT` extends type `supT`.

`Implements(subT:T,supT:T).` `Implements(subT,supT)` is true iff type `subT` is a class, type `supT` is an interface, and class `subT` implements interface `supT`.

These type related EDB predicates capture the essence of the type information available from the AspectJ shadow model. Besides the type related EDB predicates, we also have two following built-in Intensional DataBase (IDB) inference rules, which can be used to deduce whether two types have super-type relationships, and which are very useful when writing more complex queries.

`SuperType(subT:T, supT:T).` `SuperType(subT,supT)` is true iff type `subT` is a proper sub-type of type `supT`.

`SuperOrEqualType(subT:T, supT:T).` `SuperOrEqualType(subT, supT)` is true iff type `subT` is a proper sub-type of type `supT`, or `subT` is `supT` itself.

Making use of the Datalog language's transitive closure and transitive reflective closure computation capability, the two predicates can be easily implemented as the following program in Listing 4.4.

Listing 4.4: Built-in super-type deduction rules

```
SuperType(subT,supT) :- Extends(subT,supT).
SuperType(subT,supT) :- Implements(subT,supT).
SuperType(subT,supT) :- SuperType(subT,middleT),
                        SuperType(middleT,supT).

SuperOrEqualType(subT,supT) :- SuperType(subT,supT).
SuperOrEqualType(subT,supT) :- subT=supT.
```

There are a few type related predicates (and later on shadow related predicates) referring to modifiers. To query the contents of those modifiers, the user can use one or some of the following modifier related predicates, whose informal semantics are given below.

`ModIsPublic(mod:Mod).` `ModIsPublic(mod)` is true iff `mod` has the *public* modifier.

`ModIsProtected(mod:Mod).` `ModIsProtected(mod)` is true iff `mod` has the *protected* modifier.

`ModIsPrivate(mod:Mod).` `ModIsPrivate(mod)` is true iff `mod` has the *private* modifier.

`ModIsNative(mod:Mod).` `ModIsNative(mod)` is true iff `mod` has the *native* modifier.

`ModIsVolatile(mod:Mod).` `ModIsVolatile(mod)` is true iff `mod` has the *volatile* modifier.

`ModIsSynchronized(mod:Mod).` `ModIsSynchronized(mod)` is true iff `mod` has the *synchronized* modifier.

`ModIsStatic(mod:Mod).` `ModIsStatic(mod)` is true iff `mod` has the *static* modifier.

`ModIsAbstract(mod:Mod).` `ModIsAbstract(mod)` is true iff `mod` has the *abstract* modifier.

`ModIsFinal(mod:Mod).` `ModIsFinal(mod)` is true iff `mod` has the *final* modifier.

Now we turn to the design of the Datalog representation of the shadow information.

Shadow information comes from the program being woven with aspects, or called the base program, and the aspects themselves. We have designed a Datalog presentation for all of the commonly used AspectJ shadows and their static signature information.

Below are these shadow EDB predicates and their informal semantics.

`FieldGet(sh:SH,sig:Sig,parent:SH).` `FieldGet(sh,sig,parent)` is true iff `sh` is a field get shadow, whose static signature is identified as `sig`, and the shadow is lexically located within the shadow identified as `parent`.

`FieldSet(sh:SH,sig:Sig,parent:SH).` `FieldSet(sh,sig,parent)` is true iff `sh` is a field set shadow, whose static signature is identified as `sig`, and the shadow is lexically located within the shadow identified as `parent`.

`ConstructorCall(sh:SH,sig:Sig,parent:SH).` `ConstructorCall(sh,sig,parent)` is true iff `sh` is a constructor call shadow, whose static signature is identified as `sig`, and the shadow is lexically located within the shadow identified as `parent`.

`ConstructorExec(sh:SH,sig:Sig).` `ConstructorExec(sh,sig)` is true iff `sh` is a constructor execution shadow, whose static signature is identified as `sig`.

`MethodCall(sh:SH,sig:Sig,parent:SH).` `MethodCall(sh,sig,parent)` is true iff `sh` is a method call shadow, whose static signature is identified as `sig`, and the shadow is lexically located within the shadow identified as `parent`.

`MethodExec(sh:SH,sig:Sig).` `MethodExec(sh,sig)` is true iff `sh` is a method execution shadow, whose static signature is identified as `sig`.

`AdviceExec(sh:SH,sig:Sig).` `AdviceExec(sh,sig)` is true iff `sh` is an advice execution shadow, whose static signature is identified as `sig`.

`ExceptionHandler(sh:SH,ex:T,parent:SH).` `ExceptionHandler(sh,ex,parent)` is true iff `sh` is an exception handler shadow,

which catches exceptions of type `ex`, and the shadow is lexically located within the shadow identified as `parent`.

Almost all shadow predicates have a property from the domain of `Sig`, which captures the most important static signature information about the shadows. Shadow signature predicates, which will be presented below, are designed to be "layered" in a way that is similar to the inheritance in object-oriented paradigm, since Datalog itself does not directly support inheritance. The motivation of this layered design is that it is typical that at a given time, only a part of a shadow's signature information needs to be accessed, and so this design allows the programmer to only use the predicates that are needed, and thus decrease the size of the EDB that need to be fed into the Datalog solver to boost the performance of the system. For example, at a given time, for a method execution shadow, we may only want to query which class has defined this method, but not what its second parameter type is. The other benefit of this layered design is that it allows us to do more sophisticated type checking on the user written Datalog program to reject programs that do not make sense, if we want to do that. For example, a program may have qualified a shadow to be a method call shadow by using the `MethodCall` predicate, but at the same time, it tries to access a field signature information through the signature property from the method call predicate. This query is guaranteed to always return the empty result, and thus should be rejected.

The following are the shadow signature related predicates and their informal semantics.

`Signature(sig:Sig,dt:T,name:S,mod:Mod).` `Signature(sig,dt,name,mod)` is true iff `sig` is a shadow signature that has a declaring type `dt`, its corresponding program element name is `name`,

and it has modifiers identified as `mod`. Following the AspectJ's convention, the declaring type refers to the type in which the shadow member is declared.

`FieldSignature(sig:Sig,ft:T).` `FieldSignature(sig,ft)` is true iff `sig` is a field get or set shadow signature whose corresponding field's type is `ft`. Additional information about a field signature should be accessed by using a relational join operation with predicate `Signature` on the first attribute of the predicates.

`CodeSignature(sig:Sig,psig:PSig).` `CodeSignature(sig,psig)` is true iff `sig` is a constructor/method/advice shadow signature and its corresponding member's parameter type signature is identified as `psig`. Earlier, we have presented how `PSig` related predicates are defined. Additional information about a code signature should be accessed by using a relational join operation with predicate `Signature` and/or predicate `MethodSignature` (presented below) on the first attribute of the predicates.

`MethodSignature(sig:Sig,rt:T).` `MethodSignature(sig,rt)` is true iff `sig` is a method/advice shadow signature and its corresponding method/advice's return type is `rt`. Constructors do not have a return type so a signature associated with a constructor shadow does not have a `MethodSignature` layer. Additional information about a method signature should be accessed by using a relational join operation with predicate `Signature` and/or predicate `CodeSignature` on the first attribute of the predicates.

With all those shadow related EDB predicates, now we can easily write more complex queries that otherwise would be impossible to express in AspectJ's native pointcut languages. We will see plenty of

examples later, but we just give a couple of examples here to facilitate discussions.

In the JBoss AOP [32] system, after the designers realized that using the native AspectJ-like pointcut language, it was impossible to express pointcut queries that pick up method calls made on target classes that have specific fields, they came up with and added one new native pointcut *hasField* to the existing supported pointcuts. Although it is always possible to add this kind of new pointcuts to an AOP language as requirements grow, each new pointcut to be added requires advanced understanding about the compiler and how the shadow matching and weaving process works, and thus an average AOP programmer would not be able to do that. Using our approach, however, the programmer just writes a simple Datalog query to express the constraint that she would like to impose on the shadows, then our system, with the integration with the AspectJ's shadow matching and weaving process, which will be presented a little later, can automatically pick up her query results.

As an example, Listing 4.5 is a simple Datalog query program to pick up method calls made on target classes having a field named *logger*,using the EDB predicates provided by our system. The built-in predicates are highlighted with a underline for easy reading.

Listing 4.5: Pick up method calls on classes with logger field

```
WantedMethodCalls(sh) :- MethodCall(sh,sig,_),
                         Signature(sig,dt,_,_),
                         DeclaresField(dt,_,"logger",_) .
```

Of course, using our system, one can even express queries that can already be done by using AspectJ's native pointcuts, but it may not be a good idea to do so from the usability perspective. For example, using AspectJ's native pointcuts, one can use `execution(* A.compute (int))` to pick up a method definition by class `A`, whose name is

`compute` and that has only one parameter with type `int`. If one has to write the same query using our system, one would need to write a Datalog program as listed in Listing 4.6.

Listing 4.6: Datalog query to simulate a simple AspectJ pointcut

```
WantedMethodExecs(sh) :- MethodExec(sh,sig),
                         Signature(sig,t,"compute",_),
                         TypeInfo(t,"A"), CodeSignature(sig,psig),
                         CodeSignatureNumParams(psig,1),
                         CodeSignatureParam(psig,0,intT),
                         TypeInfo(intT,"int") .
```

It is obvious that in this particular case, using the AspectJ's native pointcut would be preferred. This example demonstrates that each query mechanism has its own strength, and ideally we should get advantages from both sides, with AspectJ's native pointcut language focusing on simpler syntax pattern-based queries and with our Datalog based query mechanism focusing on more complex queries that are impossible to achieve by using the native pointcuts. This actually is one of our primary design goals, and we will show how to achieve this a little later.

4.4.1.3 Static Method Call Resolution Using Shadows

To demonstrate that our Datalog based shadow EDB can indeed be used to solve non-trivial program analysis problems, we have implemented a static method call resolution algorithm based on Class Hierarchy Analysis (CHA). In programs written in Object-oriented programming languages like Java, the precise call graph is not known at compile time. One of main reasons for this is that Java-like OO languages support dynamic binding that allows a method call site to pick up a method definition to execute based on the run time type of the

target object. Since at the compile time, we do not have the precise run time type information about a target object, this posts challenges to many program analysis tasks where a whole call graph is needed. On the other hand, the static signature of a method call gives us very useful information about predicating what method definitions *may be* invoked by it at run time, and thus we can build an approximated call graph from the input program.

To achieve this, we have implemented a Datalog program, which, given a method call shadow, together with its static signature information, can deduce the set of possible method execution shadows that may potentially be invoked by it, by doing type hierarchy analysis. It essentially simulates Java's class method resolution algorithm statically, and it is very useful when we want to discover the call graph of the input program, in the presence of inheritance and polymorphism.

The key predicate of the program is `ResolvesTo(call:SH,exec:SH)` that defines on two attributes in the domain of shadow. The first attribute would be a method call shadow, and the second attribute would be a corresponding method execution shadow that may be invoked by the method call, based on the static signature information about shadows and the types. Due to its general usefulness in call graph analysis related problems, we have also incorporated it into our system as one of the built-in predicates.

There are two cases to be analyzed. The first case where a method execution can be invoked by a call is that the method execution has the same method name and the same static parameter type signature as the method call's, and the method execution (definition) is provided by any sub-type of the static type of the method call's target object. This is because, at run time, the actual run time type of the target object can be any sub-type of its static type, and thus any com-

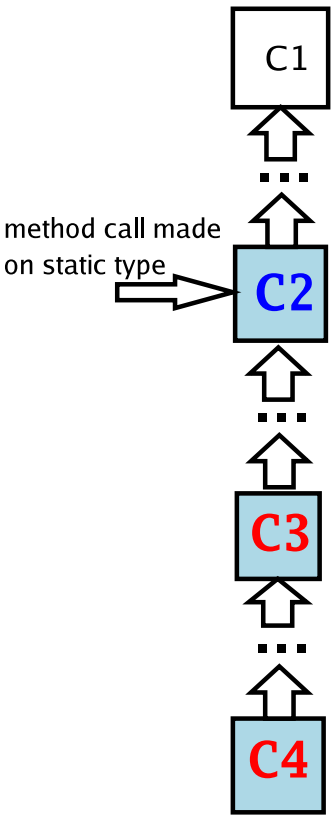

Figure 4.4: Method Resolution: case 1

patible method implementation provided by those sub-types can be invoked by the call. Figure 4.4 illustrates this scenario, and Listing 4.7 is the corresponding Datalog program to compute the `ResolvesTo` relation in this case. The `ResolvesTo` relies on two helper predicates, i.e., `CallExecMatchSig` and `TypeHasTheImpl`, to determine if one of the method call target type's sub-types has a method execution that has the signature matching with the call's.

Listing 4.7: ResolvesTo predicate: case 1

```
CallExecMatchSig(c,e,tc,te,n,psig) :- MethodCall(c,sig,_),
                                      Signature(sig,tc,n,_),
                                      CodeSignature(sig,psig),
                                      TypeHasTheImpl(te,e,n,psig) .
TypeHasTheImpl(t,e,n,psig) :- MethodExec(e,sig), Signature(sig,t,n,_),
                              CodeSignature(sig,psig) .
ResolvesTo(c,e) :- CallExecMatchSig(c,e,tc,te,_,_),
                   SuperOrEqualType(te,tc) .
```

The second case to be analyzed is a little more tricky. When one method call is made on static target type `t`, if `t` itself does not have a method implementation for it, a method execution (definition) provided by one of its super-types can be invoked. There may be more than one super-types of it that actually provide an implementation of such a method. However, only the one provided by the *closest* ancestor type can be invoked by the call, as the ones provided by the more remote ancestors are hidden by the closest one. Figure 4.5 illustrates such a scenario and Listing 4.8 is the Datalog program that actually does the analysis for this case.

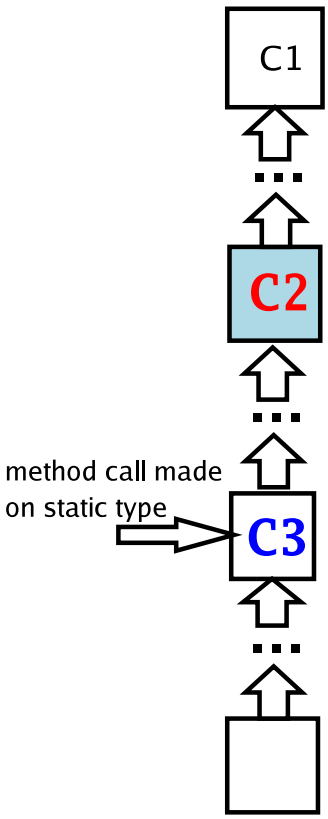

Figure 4.5: Method Resolution: case 2

Listing 4.8: ResolvesTo predicate: case 2

```
TypeHasAnImpl(t,n,psig) :- TypeHasTheImpl(t,_,n,psig) .
ThereIsCloserType(t_sofar,from_t,n,psig) :-
                                SuperType(from_t,better_t),
                                SuperType(better_t, t_sofar),
                                TypeHasAnImpl(better_t,n,psig).
IsClosestType(sup_t,curr_t,n,psig) :-
                        !ThereIsCloserType(sup_t,curr_t,n,psig),
                        SuperType(curr_t,sup_t),
                        TypeHasAnImpl(sup_t,n,psig) .
ResolvesTo(c,e) :- CallExecMatchSig(c,e,tc,te,n,psig),
                !TypeHasAnImpl(tc,n,psig),
                IsClosestType(te,tc,n,psig).
```

In a nutshell, the `ResolvesTo` predicate in Listing 4.8 makes use of three helper predicates (one of them, `CallExecMatchSig`, has been

presented earlier in Listing 4.7) to state that if a target method call type `tc` itself does not have an appropriate implementation for the method, then the call can potentially invoke the method implementation provided by the super-type that is closest to `tc` in the inheritance chain, if there is any. As a side mark, this Datalog program demonstrates one of the most visible inconveniences when programing with Datalog, i.e., one won't be able to directly express conditions quantified with *for all*, like the `IsClosestType` predicate in the example. Instead, one will have to express the negation of the condition as a new predicate first, which is the *there exists* situation where the condition does not hold, like the `ThereIsCloserType` predicate in the example, and then gets the negation of the newly created predicate to express the real intention.

4.4.2 Datalog-based Pointcuts and Integration with AspectJ

Now we describe in our system, how we can allow programmers to write pointcuts to capture design rules being interested, using Datalog-based EDB and IDB predicates, and how it is integrated with the AspectJ's native pointcut language mechanism and the compiler. As we have mentioned earlier, our design goal is that the Datalog-based pointcut system can be used together with AspectJ's native pointcut designators, using the common pointcut connectors, such as `&&`, `||`, `!`, `cflow`, and `cflowbelow`. The idea is that the user can continue to use AspectJ's native pointcuts to express simpler syntax pattern-based queries and use our Datalog based pointcuts or query mechanism to express more complex queries that are impossible to achieve by using the native pointcuts.

In our system, the user defines Datalog based pointcuts by supply-

ing one or more than one *Datalog pointcut specification* files, whose syntax is defined below in Listing 4.9 presented in the EBNF meta-language.

Listing 4.9: Syntax of Datalog specification file

```
DatalogSpecification = "Declarations" Declarations "Definitions"
                       Rules .
Declarations = List(Declaration) .
Declaration: PCDDeclaration | RuleDeclaration .
PCDDeclaration = "pointcut" <pcdName> Ident "(" <boundJPName>
                 Ident ":" "SH" ")" "." .
RuleDeclaration = <ruleName> Ident "(" List(IdentDomainPair) ")"
                 [<output> Output] "." .
Output = "output" .
IdentDomainPair = Ident ":" Domain .
Domain : SH | T | Mod | S | Z | Sig | PSig .
SH = "SH" .
T = "T" .
Mod = "Mod" .
S = "S" .
Z = "Z" .
Sig = "Sig" .
PSig = "PSig" .
Rules = List(AbstractRule) .
AbstractRule : Fact | Rule .
Fact = <factName> Ident "(" List(Literal) ")" "." .
Literal : LitNumber | LitString .
LitNumber = Integer .
LitString = String .
Rule = Head ":-" Body "." .
Head = <ruleHeadName> Ident "(" List(Ident) ")" .
Body = List(AtomicOrNegation) .
AtomicOrNegation : Atomic | NegationAtomic .
Atomic = <usedRuleName> Ident "(" List(ArgOrWildcard) ")" .
NegationAtomic = "!" AtomicOrNegation .
ArgOrWildcard : Argument | WildCard .
WildCard = "_" .
Argument : Var | Num | Str | RegExp .
Var = Ident .
Num = Integer .
```

```
Str = String .
RegExp = "REGEXP" "(" <exp> String ")" .
```

A Datalog pointcut specification file must have two sections, which are *Declarations* and *Definitions* sections respectively. The *Declarations* section should provide type declarations for the new IDB predicates that the user wants to introduce. Domains of the attributes of the predicates must be among the domains defined earlier. A predicate declaration can optionally have the *pointcut* annotation at the beginning, which will make the predicate accessible as a Datalog pointcut that can be used in an AspectJ aspect definition, and it must have and only have one attribute whose domain is the shadow domain `SH` . We also realize that in the interest of expressing design rules, not all design rule violations can be characterized as an AspectJ shadow, e.g., a violation may be the class declaration itself, which is not an AspectJ shadow. To incorporate this kind of use cases, our system allows the user to declare a Datalog predicate with the *output* annotation in the end, and this annotation will instruct the Datalog solver that the user does want to compute and output the result of the predicate, even though it is not directly accessible from an aspect. All other rules without the *pointcut* and the *output* annotations declare intermediate IDB predicates that help define a pointcut predicate or an output predicate.

The intention of the *Declarations* section is to allow us to run type checking on user supplied IDB relations to reject illegal rules early on, before they are fed into the Datalog solver. Examples of ill-typed Datalog pointcut specification files include predicates using wrong number of attributes in definitions than what have been declared, the same variable being used inconsistently on domains in a IDB relation definition, a used predicate is not declared, nor is it a built-in predicate, a declared predicate does not have a corresponding defini-

tion, and so on. Every predicate, except the built-in ones provided by the system, must be declared before it can be defined or used in the *Definitions* section.

The *Definitions* section should provide implementations of the declared IDB predicates by using the built-in ones and the newly declared ones.

Listing 4.10 is an example Datalog pointcut specification file that defines a Datalog pointcut `P` that is built on another IDB predicate `ClassHasStaticField` and some other built-in predicates in our proposed Datalog-based approach. In the example, pointcut `P` selects any method call shadow whose target type has a static field. This kind of pointcut selection is not possible when using the native AspectJ pointcut designators.

Listing 4.10: An Datalog pointcut example

```
Declarations
 pointcut P(s:SH) .
 ClassHasStaticField(t:T) .
Definitions
 ClassHasStaticField(t) :- IsClass(t),DeclaresField(t,_,_,mod),
                           ModIsStatic(mod) .
 P(s) :- MethodCall(s,sig,_),Signature(sig,tt,_,_),
         ClassHasStaticField(tt) .
```

Once a Datalog-based pointcut has been declared and implemented in a pointcut specification file, in our system, it then can be used in a regular aspect, just like native pointcut designators, but with a little special syntax. Listing 4.11 is an example aspect that uses the previously defined Datalog pointcut `P`.

Listing 4.11: An aspect using Datalog pointcut

```
aspect Example {
 pointcut capture(): call(* *.run()) && ?P();
 declare error: capture() :
       "A run method is called on a class
          with a static field";
 }
}
```

The special symbol `?` preceding `P` in Listing 4.11 indicates the pointcut, `P`, is a Datalog-based pointcut so that during the pointcut evaluation stage in our enhanced AspectJ compiler, the set of shadows that are bound to the only attribute of `P` will be the result of the evaluation of `?P()`, which is further refined by the conjunction of the native AspectJ pointcut expression, as shown in the example. The pointcut `capture()` in the aspect in this example will find method calls made to any method whose name is `run` and whose target type is a class with at least one static field, and when it finds one, the AspectJ compiler will report an error with the associated shadow lexical information.

Of course, when compiling this aspect, the user has to supply the Datalog specification file, the aspect, and the base program to the enhanced AspectJ compiler for compiling, weaving, and/or error reporting.

4.4.2.1 High-level overview of the Implementation of the Datalog pointcut system

We have implemented the Datalog based pointcut evaluation system by enhancing the industrial standard Eclipse AspectJ compiler (version 1.5) and by making use of the Datalog solving functionality provided by the third party Datalog solver *bddbddb* that can leverage the

power of BDD.

Figure 4.6 illustrates a high-level overview of the major components in the implementation of our Datalog based pointcut system, and each component's role in the whole AspectJ compiler's compilation and weaving process. In the figure, the boxes with boldface borders are new components that are added to the Eclipse AspectJ compiler to address the need of bringing Datalog query capability to the pointcut matching mechanism. The functionalities of those new components are briefly explained below.

- Type checking. This component takes in a user-supplied Datalog pointcut specification file and checks whether the specification is well-typed. This includes making sure that: all user-defined Datalog predicates are declared before being used, that the usages of the user-defined predicates or the EDB/built-in IDB predicates are consistent with their declarations in terms of number of variables and the domains of the variables, and that variables occurring more than once in a rule have consistent domains.

- Dependency analysis. This component scans through the user-supplied Datalog pointcut specification file to try to determine what EDB predicates or built-in IDB predicates need to be included in the input fed into the Datalog solver. Irrelevant EDB/built-in IDB predicates will be pruned to speed up the Datalog solving.

- Create EDB/IDB. This component takes in a collection of weavable shadows from the earlier stage of the weaving, and the dependency information got from the previous step, then it generates the EDB and IDB for solving. Meanwhile, this component

maintains the mappings between the shadows known to the AspectJ compiler and the corresponding Datalog objects defined in the shadow domain in the generated EDB, so that later on we can compose the solved shadow results from the Datalog solver back into the AspectJ weaving process.

- Feed into BDDBDDB. This component feeds the generated EDB/IDB to the Datalog solver, *bddbddb*, to solve the constraints to get the result shadows or output predicates.

- Compose. This component gets the resolved shadows from the previous step, using the mapping information stored earlier, and evaluate the composed pointcut expressions (including both regular pointcuts and Datalog pointcuts) to get the final pointcut matching results for weaving or reporting design rule violations.

4.5 Case Studies

We have carried out a few case studies to evaluate the effectiveness of our Datalog based pointcut system when being used to detect design rule violations. Here we present three of them that we think can best demonstrate the salient features of our system. More extensive evaluation can be found in the evaluation chapter, Chapter 5.

4.5.1 Case Study 1: Java hashCode/equals Methods Rule Checking

As we have explained in the introduction chapter, there are two important Java design rules with regard to how to override `Object` class's `hashCode` and `equals` methods in a derived class. The design rules state that:

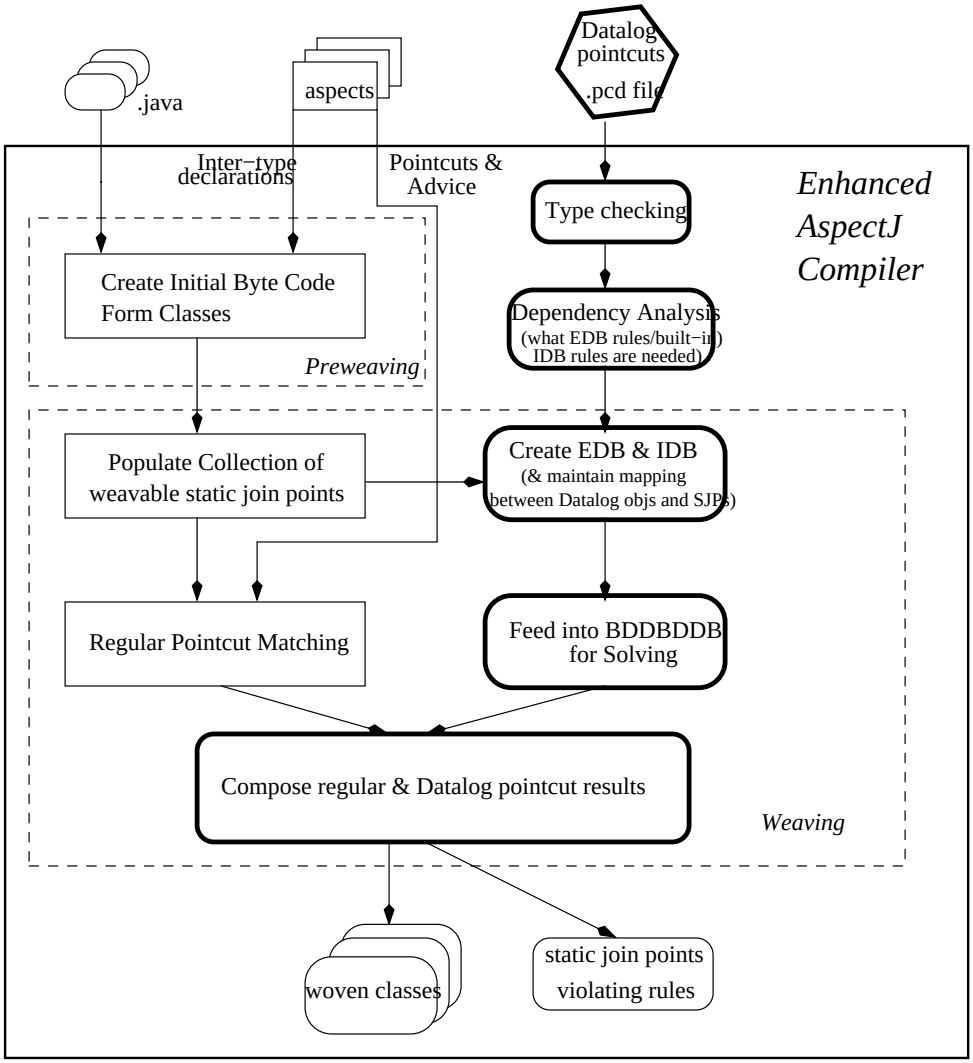

Figure 4.6: Overview of the Components of the Datalog pointcut system and their roles in the AspectJ weaving

- In a Java class, when one overrides the `equals` method, one should override the `hashCode` method as well, and
- when one does so, one should make sure the two methods are using the same set of fields.

The design rules can be easily expressed using our Datalog-based pointcut mechanism. The pointcut specification file in Listing 4.12 defines two pointcuts, *EqualsHasNoHashcode* and *HashcodeEqualsUseDiffFields*. The aspect defined in Listing 4.13 uses the two Datalog

pointcuts in conjunction with a native pointcut, which selects all `equals` execution shadows, to catch any violation of the two design rules in the base program respectively.

Listing 4.12: Datalog pointcuts for equals/hashCode Design Rules

```
Declarations
  pointcut EqualsHasNoHashcode(eq:SH) .
  EqualsHasHashcode(eq:SH,hc:SH) .
  pointcut HashcodeEqualsUseDiffFields(eq:SH) .
  EqualsUsesField(eq:SH, f:Sig) .
  HashcodeUsesField(eq:SH, f:Sig) .
Definitions
  EqualsHasHashcode(eq,hc) :- MethodExec(eq,s), Signature(s,t,_,_),
                          MethodExec(hc,s2),Signature(s2,t,"hashCode",_) .
  EqualsHasNoHashcode(eq) :- MethodExec(eq,_),!EqualsHasHashcode(eq,_) .

  EqualsUsesField(eq,f) :- FieldGet(_,f,eq) .
  HashcodeUsesField(eq,f) :- EqualsHasHashcode(eq,hc),FieldGet(_,f,hc) .
  HashcodeEqualsUseDiffFields(eq) :- EqualsUsesField(eq,f),
                                  !HashcodeUsesField(eq,f),
                                  EqualsHasHashcode(eq,_) .
  HashcodeEqualsUseDiffFields(eq) :- EqualsHasHashcode(eq,_),
                                  !EqualsUsesField(eq,f),
                                  HashcodeUsesField(eq,f) .
```

Listing 4.13: Aspect for Enforcing equals/hashCode Design Rules

```
aspect HashEqChecker {
    pointcut p1(): execution(boolean *.equals(Object)) &&
                   ?EqualsHasNoHashcode() ;
    declare error: p1() :
            "Equals method has no corresponding hashCode method!";

    pointcut p2(): execution(boolean *.equals(Object)) &&
                   ?HashcodeEqualsUseDiffFields() ;
    declare error: p2() :
      "Hashcode and Equals should have used the same set of fields!";
}
```

In Listing 4.12, pointcut `EqualsHasNoHashcode` selects any class's `equals` method implementation that has no accompanying `hashCode`

implementation. Note that in the pointcut Datalog implementation itself, we do not actually explicitly specify the selected shadow must be an `equals` method. This is because we know the pointcut will be further refined when used in conjunction with the native pointcut in the aspect, which only selects an `equals` method. This unique feature of our system sometimes simplifies a Datalog pointcut implementation since the native AspectJ pointcuts are good at picking up program shadows based on syntax patterns.

Pointcut `HashcodeEqualsUseDiffFields` finds any class's `equals` method implementation that uses a different set of fields than what is used by its counterpart `hashCode` method, when both are present. Again, in this pointcut Datalog implementation, we do not explicitly specify the result shadow to be an `equals` method and leave it to the native pointcut in the aspect. The overall Datalog implementation of the two pointcuts is very succinct and is easy to understand.

4.5.2 Case Study 2: Law of Demeter Static Checker

As we have noted about our statically executable advice based Law of Demeter (LoD) static checker implementation, it has at least two problems: 1. It relies on an assumed order in which the program shadows are visited, i.e., a method call execution shadow must be visited by the aspect weaver before a method call shadow contained in it is visited, which is not guaranteed to always be true for another compiler implementation; 2. it is difficult to connect a LoD sub-rule with the corresponding part of the implementation.

Thanks to the declarativeness of the Datalog language and our Datalog based shadow representation, we are able to achieve a better implementation of a LoD static checker for the class form LoD, as listed in Listing 4.14. We omit the aspect that uses the Datalog point,

as it simply just declares a `declare error` statement on the Datalog pointcut and prints out an error message.

Listing 4.14: LoD checker in Datalog pointcuts

```
Declarations
pointcut LoDViolateCall(c:SH) .
LoDConform(c:SH) .
CallWithinClass(c:SH,t:T) .
CallWithTargetType(c:SH,t:T) .
ClassHasFieldType(t:T,ft:T) .
ContextHasArgType(exe:SH, at:T) .
ContextUsesConstruType(exe:SH, ct:T) .
ContextUsesReturnType(exe:SH, ct:T) .

Definitions
LoDViolateCall(c) :- MethodCall(c,_,_), !LoDConform(c) .

CallWithTargetType(c,t) :- MethodCall(c,sig,_),Signature(sig,t,_,_) .

CallWithinClass(c,t) :- MethodCall(c,_,caller),
                        MethodExec(caller,sig),
                        Signature(sig,t,_,_) .
LoDConform(c) :- CallWithTargetType(c,t), CallWithinClass(c,t) .

ClassHasFieldType(t,ft) :- DeclaresField(t,ft,_,_) .
LoDConform(c) :- CallWithTargetType(c,ct), CallWithinClass(c,t),
                 ClassHasFieldType(t,ct) .

ContextHasArgType(exe,at) :- MethodExec(exe,sig),
                             CodeSignature(sig,csig),
                             CodeSignatureParam(csig,_,at) .
LoDConform(c) :- CallWithTargetType(c,t), MethodCall(c,_,caller),
                 ContextHasArgType(caller,t) .

ContextUsesConstruType(exec,ct) :- ConstructorCall(_,sig,exec),
                                   Signature(sig,ct,_,_) .
LoDConform(c) :- CallWithTargetType(c,t), MethodCall(c,_,caller),
                 ContextUsesConstruType(caller,t) .
```

```
36 ContextUsesReturnType(exec,rt) :- MethodCall(_,sig,exec),
                                     MethodSignature(sig,rt) .
38 LoDConform(c) :- CallWithTargetType(c,t), MethodCall(c,_,caller),
                    ContextUsesReturnType(caller,t) .
```

Inside the pointcut specification file, the pointcut predicate `LoDConform` is the key predicate that captures the five cases where a target type is allowed by the LoD: the hosting `this` class itself, the field types of the `this` class, the argument types of the enclosing method body, the classes whose constructors have been used by the enclosing method body, and the return types of the used method calls within the enclosing method body. The key advantage of this implementation can be seen by noting the five inference rules of the `LoDConform` predicate (it appears five times as a rule head in the file) have the exact mappings to the corresponding five cases of the LoD definition. Furthermore, there is no need to assume any traversal order or evaluation order of shadows, so the implementation is still correct if we have a different implementation strategy of the AspectJ compiler.

4.5.3 Case Study 3: Detect Recursive Calls in Presence of Polymorphism and Aspects

Recursive call chain is not always allowed in programs. For example, in embedded systems, recursion generally is disallowed in production code, due to safety requirements.

Detecting recursive call chain in programs written in object-oriented programming languages is challenging because of the polymorphism. The presence of AOP aspects further complicates the issue because they can change the call graph in implicit ways. This case study of writing a static checker to report recursive call chains in AOP programs is also motivated by a real bug we encountered when imple-

menting an AspectJ based dynamic checker for the object form of the LoD as a part of work for [34]. The bug was so subtle that it took us a few days before we eventually figured out what the issue was. According to our experience, this kind of bugs are not uncommon when programming in AspectJ.

To understand what the problem was, Listing 4.15 is a simplified version of our aspect that tries to dynamically detect the object form of LoD violations in the program execution. The aspect advises executions of all methods by maintaining a stack that stores the allowed target objects in each execution context. It also advises all method calls to check if the target objects are among the allowed ones in the context, and then reports a run time error if the target object is not an allowed one.

Listing 4.15: Dynamic LoD Checker Aspect

```
aspect SimpleLoDChecker {
 java.util.Stack stacks = new java.util.Stack();

 before(): execution(* *.*(..)) && !within(SimpleLoDChecker) {
  java.util.HashSet set = new java.util.HashSet();
  stacks.push(set);
  set.add(thisJoinPoint.getTarget());
 }

 after(): execution(* *.*(..)) && !within(SimpleLoDChecker) {
  stacks.pop();
 }

 before(): call(* *.*(..)) && !within(SimpleLoDChecker) {
  java.util.HashSet set = (java.util.HashSet)stacks.peek();
  if(!set.contains(thisJoinPoint.getTarget()))
    System.out.println("Violating LoD!");
 }
}
```

When writing aspect code, one commonly made mistake is that an aspect inadvertently advises the code of the aspect itself, and thus creates an infinite call chain in the call graph. We indeed were aware of this issue when writing the aspect, as evident from the fact that all of the pointcut expressions have a conjunction condition `!within (SimpleLoDChecker)`, which instructs the AspectJ compiler to not to insert advice to shadows occurring in the aspect itself. Surprisingly, when we applied this version of the aspect to one of the test benchmarks, the woven code still hit an infinite recursive call chain and ran out of stack space. When we ran the test benchmark alone, without the aspect, the problem went away. Even more surprisingly, for all other test benchmarks, there was no such an issue. We wished we had a tool to help us identify which advice had caused such a problem, as it took us a few days to eventually figure out what went wrong.

Using our Datalog based system, we are now able to write a static checker to report which method calls within the aspect can potentially create a recursive call chains, in the presence of inheritance, polymorphism, and aspects. Listing 4.16 is our Datalog implementation of such a checker.

Listing 4.16: Datalog Checker for Recursive Call Chain

```
Declarations
pointcut LoopStartingFrom(jp:SH) .
MaybeReachableByShadow(jpTo:SH, jpFrom: SH) .
MaybeReachableByShadowBase(jpTo:SH, jpFrom: SH) .
AspectLoop(t: T) .
Definitions
MaybeReachableByShadowBase(jpTo, jpFrom) :- MethodExec(jpFrom,_),
                                            MethodCall(jpTo,_,jpFrom) .
MaybeReachableByShadowBase(jpTo, jpFrom) :- AdviceExec(jpFrom,_),
                                            MethodCall(jpTo,_,jpFrom) .
MaybeReachableByShadowBase(jpTo, jpFrom) :- MethodCall(jpFrom,_,_),
                                            ResolvesTo(jpFrom,jpTo),
                                            MethodExec(jpTo,_) .
MaybeReachableByShadowBase(jpTo, jpFrom) :- MethodCall(jpFrom,sig,_),
                                            AdviceExec(jpTo,sig) .
MaybeReachableByShadow(jpTo, jpFrom) :-
                              MaybeReachableByShadowBase(jpTo,jpFrom) .
MaybeReachableByShadow(jpTo, jpFrom) :-
                              MaybeReachableByShadowBase(jpTo,jpTo1),
                              MaybeReachableByShadow(jpTo1,jpFrom).
AspectLoop(t) :- IsAspect(t), TypeInfo(t,"SimpleLoDChecker")

LoopStartingFrom(from) :- MethodCall(from,_,theExec),
                          AspectLoop(aspT),
                          AdviceExec(theExec,sig2),
                          Signature(sig2,aspT,_,_),
                          MaybeReachableByShadow(to,from),
                          from=to .
```

Datalog pointcut `LoopStartingFrom` returns the result method call shadows within the aspect that may potentially create a recursive call chain in the woven code. It relies on a helper predicate `MaybeReachableByShadow` to calculate the transitive closure of the reachability relationship between method calls, method executions, and advice executions, by

applying the base case predicate `MaybeReachableByShadowBase`. The base case predicate defines four cases where one of those method-/advice shadows can reach another, with one of them utilizing our `ResolvesTo` predicate that statically approximate the set of method executions that may be invoked by a method call.

Using this static checker on the woven code for the problematic benchmark, we are able to identify the causing method call in the aspect, given by the result returned by the Datalog pointcut `LoopStartingFrom`. It is method call `set.add` in the aspect that is the source of the infinite call chain. It turns out that in that test benchmark, there is one class that overrides the `hashCode` method, which is being advised by the `before` advice on method execution join points. The `set.add` method within the advice, however, needs to call the argument object's `hashCode` method to calculate the object's hash value, which thus results in an infinite call chain. So in the original aspect definition, the `!within(SimpleLoDChecker)` pointcut is not strong enough to prevent this from happening, as the aspect indeed did not advise the code of itself. To fix this problem, we should use `!cflow (within(SimpleLoDChecker))` as the conjunction condition in each of the pointcut definitions in the original aspect.

CHAPTER 5

Evaluation

We carry out a few experiments to evaluate our approach. The evaluation is focused on the following three aspects of our approach and the implementation: its effectiveness of describing software design rules and detecting violations, its usability by comparing it with an alternative approach, and its performance and scalability with regard to program sizes.

5.1 Effectiveness evaluation

The effectiveness evaluation is carried out by implementing and analyzing the software design rule sets coming from two prominent Object-oriented software static checkers, which are Microsoft's *FxCop* [16] and an open source Java bug finding tool called *FindBugs* [27, 53]. The goal is to find out that among the identified design rules, what rules can be captured by our approach and tool, and what cannot be. Through this process, we also wish to know if there is any lesson that can be learned from implementing those design rules.

5.1.1 Implement FxCop Design Rules

FxCop is Microsoft's .NET framework code assembly static analyzing tool. It checks and reports information about a code assembly, such as

possible design, localization, performance, and security issues [16]. Besides the rule checkers shipped with FxCop by default, it allows advanced programmers to use complicated .NET introspection API to write their customized rule checkers. According to the 1.35 version we have today, the default rule checkers check nine categories of software issues. In this particular evaluation, we are only interested in the category of *Design Rules*, as it is most relevant to the work being presented here.

FxCop has 60 design rules, among which there are 32 rules that are concerned with .NET specific features, and so those rules are not applicable in this study. After analyzing the rest of 28 rules, including actually implementing 10 of them, we conclude that 25 of them can be implemented using our aspect shadow model based approach, with pretty low implementation effort for each. Among the 3 rules that we are not able to capture, 2 of them are due to the aspect shadow model is not expressive enough, and the remaining one is due to the design rule description itself is not clear enough to make the determination. At the end of the section, we have a table summarizing why each of the 2 rules cannot be implemented using our approach.

Among the 10 design rules that we have actually implemented, most of them only need to use a few Datalog literals, thanks to the shadow model and the declarative nature of the Datalog language. For the rest of the other 15 rules that we did not actually implement, we did analysis on each rule, and the conclusion was that they were more aligned with the easier ones among the 10 implemented rules, and going ahead to actually implement them won't provide much more insights to us. Table 5.1 summarizes the 10 rules that we have implemented and the number of literals needed for each implementation. We use the total number of literals appearing on the right hand

sides of inference rules as a measure of the complexity of a Datalog query.

Rule abstract	Number of literals used
Abstract types should not have constructors	6
There should be no empty interface	3
Avoid excessive parameters on generic types	4
Do not catch general exceptions	6
Avoid having static members in a generic type	8
Consider passing base types as parameters	20
Avoid having protected members in a final class	5
Exceptions should be public	10
Avoid having visible instance fields	5
Do not hide base class methods	18

Table 5.1: Implemented design rules and number of literals needed

Two of the rule implementations are relatively more interesting than others. So we explain them a little here. For the rest of the eight rule implementations, interested readers can find them in Section 8.1 of the Appendix.

5.1.1.1 Case study: Consider passing base types as parameters

FxCop has this design rule to encourage wider reuse of an API method. The idea is that *using base types as parameters to methods improves reuse of these methods if you only use methods and properties from the parameter's base class* [16]. Both FxCop and FindBugs use this kind of informal statements to describe desirable design rules or programming styles. They can be better understood via examples. Below is a Java example to illustrate the particular design rule in question here:

Listing 5.1: A reusable Java API method example

```
class Foo {
  public void API(Vector v) {
    v.add(this);
  }
}
```

In Listing 5.1, the method named `API` takes in a parameter of type `Vector`, where it could have taken in a parameter of interface `Collection`, given the `add` method is the only method called inside the method body and it is being declared in the type of `Collection`, which is a super type of `Vector`. Had the programmer done so, the `API` method could have been used in more contexts, rather than just in the context of a `Vector` parameter.

Listing 5.2 is our implementation to identify all method definitions whose one or more parameters could have used more abstract types. The essence of the algorithm is represented in the predicate `TypeCantBeMoreAbstract` which has two inference rules at line 20 and line 23 respectively. Inference rule at line 20 says that a parameter type `t` for a method execution join point `jp` cannot be made more abstract, because the method execution makes a call to a method on `t`, and the latter method is not declared in any of the super types of `t`. So using the specific type of `t` is justified in such a case. Inference rule at line 23 is similar except it tries to identify such a scenario in fields accesses. Then the main output predicate `MethodShouldUseBase` just uses the negation of predicate `TypeCantBeMoreAbstract` and a few other helper predicates to identify violating method definitions.

Listing 5.2: Identify methods that should have used base class as a parameter

```
Declarations
#Domains: SH shadows
# T types
# S strings/identifiers
# Sig shadow signatures
# PSig method/constructor parameter signatures
pointcut MethodShouldUseBase(jp:SH) .
MethodParamTypes(j:SH,ps:PSig,t:T) .
TypeCantBeMoreAbstract(j:SH,t:T) .
SuperTypeHasMDeclaration(t:T,n:S,psig:PSig) .
FieldAccess(jp:SH,f:Sig) .
ClassOrInterface(t:T) .
Definitions
MethodShouldUseBase(jp) :- MethodParamTypes(jp,_,pt),
                       ClassOrInterface(pt),
                       !TypeCantBeMoreAbstract(jp,pt).
ClassOrInterface(t) :- IsClass(t) .
ClassOrInterface(t) :- IsInterface(t) .
MethodParamTypes(jp,psig,pt) :- MethodExec(jp,sig),
                             CodeSignature(sig,psig),
                             CodeSignatureParam(psig,_,pt) .
TypeCantBeMoreAbstract(jp,t) :- MethodExec(jp,_),MethodCall(_,sig,jp),
                           Signature(sig,t,n,_),
                           CodeSignature(sig,psig),
                           !SuperTypeHasMDeclaration(t,n,psig) .
TypeCantBeMoreAbstract(jp,t) :- MethodExec(jp,_),FieldAccess(jp,f),
                             Signature(f,t,_,_).
SuperTypeHasMDeclaration(t,n,psig) :- SuperType(t,tsup),
                                   DeclaresMethod(tsup,_,n,_,psig) .
FieldAccess(jp,f) :- FieldGet(_,f,jp) .
FieldAccess(jp,f) :- FieldSet(_,f,jp) .
```

While the implementation is declarative and succinct, it is worthwhile to point out one important limitation of our approach, which

is also a limitation shared with the FxCop tool. Since our approach is based on the shadow model, it is object insensitive in nature. For example, in the inference rule at line 20, the rule does not and cannot distinguish a method call on type `t` with the target object being one of the parameters or an alias of one of the parameters, and a method call also on type `t`, but with the target object being an object created locally. In the latter case, the parameter's type could still be made more abstract and thus could promote more reusability. Being able to distinguish the two cases would identify more reuse opportunities but requires more semantics based data flow analysis. Our approach is conservative in the sense when it finds a violation, then it must have been a true violation, however, it may miss violations that otherwise would be picked up by using object sensitive analyzing techniques. The same limitation applies to the FxCop tool.

5.1.1.2 Case study: Do not hide base class methods

This design rule describes a common programming error when overriding a method that has been defined by a super class. A classical corresponding example in the Java world in Listing 5.3 illustrates the problem.

Listing 5.3: A common method overriding mistake

```
class Bar {
  public boolean equals(Bar obj) {
    // equals logic
  }
}
```

The programmer intends to provide a `Bar` specific equality comparison logic by providing its own `equals` method, which is common and justified. The problem is, however, that the `equals` method here is defined on the parameter of type `Bar`, which will hide the `equals` method defined on class `Object`, because the latter takes in a parameter of type `Object`. The consequence is that when you put a `Bar` object into a hash table based collection, you will never be able to retrieve it back, since the collection uses `Object`'s `equals` method to determine the equality of two objects. The `equals` method defined in `Bar` fails to override `Object` class's `equals`, and it should have been defined on a parameter of type `Object` instead.

Listing 5.4 is our implementation to identify such kind of violating methods.

Listing 5.4: Identify methods that hide base class' methods

```
Declarations
#Domains: SH shadows
# T types
# PSig method/constructor parameter signatures
pointcut MethodHidesBase(jp:SH) .
IncompatibleType(t1:T,t2:T) .
InCompatibleSignature(s1:PSig,s2:PSig) .
Definitions
MethodHidesBase(jp) :- MethodExec(jp,sig), Signature(sig,t,n,_),
            CodeSignature(sig,psig), CodeSignatureNumParams(psig,num),
            SuperType(t,tsup), DeclaresMethod(tsup,_,n,_,psig2),
            CodeSignatureNumParams(psig2,num),
            !InCompatibleSignature(psig,psig2),
            CodeSignatureParam(psig,pos,pt1),
            CodeSignatureParam(psig2,pos,pt2),
            SuperType(pt1,pt2) .
IncompatibleType(t1,t2) :- TypeInfo(t1,_),TypeInfo(t2,_),
                  !SuperOrEqualType(t1,t2),!SuperOrEqualType(t2,t1) .
InCompatibleSignature(s1,s2) :- CodeSignatureParam(s1,pos,pt1),
                                CodeSignatureParam(s2,pos,pt2),
                                IncompatibleType(pt1,pt2) .
```

Note the usage of the negation of predicate `IncompatiableSignature` at line 13. This is needed to avoid mistakenly picking up a completely irrelevant method in a subclass, just because the method happens to have used the same method name as its super-class' method, and to have had one parameter' type be a subtype of the same parameter of the super-class' method. Although the implementation is still short enough, we wish we could avoid using this kind of negation. According to our experience, this is one of the most visible inconvenience when programing with Datalog, i.e., you won't be able to directly express conditions quantified with *for all*. Instead, one will have to express the negation of the condition as a new predicate first, which

is the *there exists* situation where the condition does not hold, and then use the negation of this newly created predicate.

5.1.1.3 Rules that cannot be expressed

As mentioned earlier, among the 28 rules, there were 2 rules that could not be expressed using our approach. Table 5.2 is the table summarizing the two rules and why they could not be expressed.

Rule abstract	Reason that it cannot be implemented
Generic method's type parameter should be inferable from its parameters' types.	Shadow lacks generic methods' type parameter information, which is essential to implement this rule. In AspectJ, shadow/pointcut matching is based only on information after the type erasure process that gets rid of type parameters.
All reference arguments passed to public methods should be tested against null before usage.	Shadow does not have runtime object/data value information, nor does it have full control flow information.

Table 5.2: Rules that cannot be expressed using our approach and why

The first row is about a commonly recognized programming practice when using generic methods in both Java and C#. A generic method has one or more than one type parameters that serve as type place holders and that can be instantiated to actual types when the method is invoked. If the type parameters are inferable from the method's formal argument types, then the user does not bear the burden of having to pass in actual type arguments when invoking the method, as the compiler is able to automatically infer them from the method arguments. Otherwise, the user will have to use a special syntax to supply actual type arguments in each place where the method is invoked.

The second row in the table is consistent with what we have explained in the section of discussing the implementation of "Consider passing base types as parameters". In cases where object sensitivity and flow sensitivity is important, we will need to rely on some other data flow analysis based tools to find programming errors.

5.1.2 How hard is it to extend shadow model?

It is natural to ask how hard it would be if one has to extend the shadow model so that it contains information useful for solving a particular program analysis problem or for solving other computation problems.

First of all, extending the shadow model is generally a challenging task. It requires a careful design about how to present the structure of the new shadow kind, what context information should be exposed and how it affects the pointcut expression language and the join point reflection API. On top of that, to implement such an extended shadow model, one also needs to be an expert on the aspect compiler architecture, understanding the shadow matching process and being proficient with library APIs dealing with low level language syntax elements such as types, control flow graphs, Java bytecodes, and etc. Just as a concrete example, it takes an AOSD paper [25] to explain how to add the loop shadow to AspectJ's shadow model, with many subtle cases having to be considered both at the design level and the implementation level. In fact, the work presented in that paper becomes the central part of the first author's PhD dissertation [24].

Now let's go back to the two concrete cases presented in Table 5.2, where a shadow model based checker could not be implemented due to the lack of information. The second case would require the shadow model to have access to information that is only available at runtime,

i.e., method parameter objects and `null`, and thus it does not make sense to add them to the shadow model that is static by nature. However, it would be possible to extend the shadow model for the purpose of implementing a checker for the first case. More specifically, at the design level, such a extension does not pose much challenge, while at the implementation level, this requires shadows be created at an even earlier stage where type parameter information is still present, before the type erasure process has occurred. Although possible to do so, this implementation change would be a dramatic change to the compiler/weaver architecture, and thus calls for a non-trivial effort.

5.1.2.1 Conclusions from the experiment

This effectiveness evaluation experiment shows that our approach is very effective for enforcing the FxCop design rule set. In particular, 25 of the 28 applicable FxCop design rules can be enforced in our approach, giving us a success rate of 89.3%. Furthermore, using our approach, each design rule checker only requires a low implementation effort and thus the tool is very usable.

5.1.3 Implement FindBugs Design Rules

FindBugs [27, 53] is a popular bug finding tool among Java programmers. According to its web site, as of July of 2008, it had been downloaded more than 700,000 times. It has more than 350 bug patterns that are categorized into nine categories. Besides, users can write their customized bug pattern detection algorithms by using a Visitor pattern approach by traversing the Abstract Syntax Tree.

Among the nine provided bug pattern categories, the most relevant category to this study is the *Bad practice* category, which has 84

bug patterns that it can detect. We pick up the first 20 bug patterns, with some of almost identical ones combined, to try to implement them to see how many of them can be implemented using our approach.

After experiments, we find out that among the picked up 20 bug patterns, we could implement and have actually implemented bug detection analyzers for 11 of them. Table 5.3 summarizes what bug patterns can be detected using our approach and what cannot and why.

Pattern abstract	Reason that it cannot be implemented 'Y' if it can be implemented
Empty jar file entry	Lack of information in shadow model
Equals method should return false if argument is not the type of *this*.	Lack of control flow and object specific information in shadow model
Random object created and used only once	Same as above
Check for sign of bitwise operations	Bitwise operation expression is not part of shadow model
Class implements *Cloneable* but does not define *clone* method	Y
clone method does not call *super.clone()*	Y
Class defines *clone* without implementing *Clonneable*	Y
Class defines covariant *comapreTo*	Y
Method might drop exception	Y
Method might ignore exception	Y
Don't use *removeAll* to clear a collection	Y
Some methods should only be invoked inside *doPrivileged* block	Lack of information in shadow model
Method invokes dangerous methods on *System* class such as *System.exit(...)*	Y
String comparison using == or *!=*	No operator usage information present in shadow model.
Equals checks for noncompatible operand	*instanceof* operator is not present in shadow model.
Class defines *compareTo* but uses *Object.equals()*	Y
equals method should fail for subtypes	No == operator usage information present in shadow model.
Covariant *equals()* method defined	Y
Empty finalizer should be deleted	Could only tell if a method does not have any shadow; could not tell if it does not have anything
Explicit invocation of *finalizer*	Y

Table 5.3: How FindBugs bug patterns can be implemented in our approach

For the 11 bug patterns that can be implemented using our approach, the implementations do not require much effort. Interested readers can find the source code of those implementations in Sec-

tion 8.2 in the Appendix.

For the remaining 9 bug patterns that cannot be implemented using our approach, the majority of them are due to some particular information not being present in the shadow model. For example, those built-in operator expressions are not weavable in the current AspectJ programming model, and so they are not part of the shadow model. It is not clear at this moment if the AspectJ language will make these kinds of expressions weavable or not in the future, but if it does, the shadow model must be extended too, and at that time, our approach can be applied to those currently unsupported cases too.

5.1.3.1 Conclusions from the experiment

This effectiveness experiment shows that for detecting bug patterns in FindBugs' pattern set, our approach is not as effective as it is for enforcing the FxCop design rule set. But among the 20 selected bad practice bug patterns, 11 of them still can be detected using our approach. That gives us a 55% success rate. For the ones that can be detected in our approach, the needed implementation effort is low. For the ones that our approach cannot implement, majority of them are due to the lack of particular information in the shadow model. Unfortunately, as discussed earlier, to enhance the shadow model to contain more information is a non-trivial task, as it requires a careful design of the proposed new shadow structure and the context information that it should expose, a thorough understanding of the underlying AOP compiler implementation, and the low level library APIs dealing with base language constructs.

5.2 Usability evaluation

We carry out an experiment to evaluate the usability of our approach. The experiment tries to answer this question: Since one can build another similar system that is not based on AOP's shadow model, e.g., by basing on the Abstract Syntax Tree model, is our model better or worse when compared with the alternative model?

Given the thesis statement argues for using AOP's shadow model as the meta model for query, to support this thesis, we need to keep the query mechanism, i.e., Datalog, unchanged, while changing the underlying model. So in this experiment, we have prototyped another Abstract Syntax Tree (AST) based system, where the AST extracted from a Java program can be converted into a Datalog Extensional DataBase (EDB), and then problem specific Datalog programs can be written against this EDB to find design rule violations in the underlying Java program. We choose AST as the alternative meta model because it is widely used in program analysis tasks.

The first step of the experiment is to build a system to convert the AST from a Java program into an EDB. It turns out that this task is more difficult than it appears to be. Initially we started with using a Demeter style approach traversing a parsed Java syntax tree and emitting the EDB during the traversal. But we soon realized that naïvely doing a simple traversal won't work. The main challenge comes from the fact that the type information is not immediately available from a syntax tree, and without the type information, it is almost useless. Examples of those type information include but not limited to: when a variable is accessed, what type of this variable is, is it a local variable, or a field (instance or static), and if it is a field, which class declares it; when a method is invoked, what method this invocation is calling, given the static information of the

call site. Getting those kind of information ready means one has to mimic what a Java compiler type checking is doing, which requires huge implementation effort and even if one does that, it is unlikely he or she can get everything right. Just as an example, it takes the Java language specification [20] a few pages just to explain the algorithm to determine which method should be called by an invocation, with many cases having to be considered.

After some research, we decided to build up this prototype system based up on an Eclipse plugin, called AST View [45], which in turn is based on the Eclipse Java Development Tools (JDT) [60]. The nice thing about this AST View tool is that it can extract AST from a Java compilation unit, and also attach the relevant type information, called *bindings*, to the corresponding AST nodes, by leveraging the deep type analysis capability provided by JDT. Bindings can be shared among different AST nodes, if several nodes happen to have the same type. So strictly speaking, this AST is no longer a tree.

We start with this AST, and then convert it into a Datalog EDB representation. Algorithm 5.1 below is our recursive algorithm that does this conversion. To do the actual conversion, one just need to call `convert(root)`, where `root` is the root node of an AST.

Algorithm 5.1: Convert (N) (N is an AST node)

```
id ← GetID(N)
predicateName ← Label(N)
fields ← [id]
for all p in properties of N do
  {for a property representing a binding, p will be the corresponding binding id}
  append p to fields
end for
insert predicate predicateName(fields) into EDB
for all e in outgoing edges from N do
  edgePredicateName ← Label(N) __ Label(e)
  if e is a repetitive edge then
    num ← Arrity(e)
    for all c in child end of e do
      insert predicate edgePredicateName(id,Position(c),GetID(c),num)
      into EDB
      convert(c)
    end for
  else
    c ← child end of e
    insert predicate edgePredicateName(id,GetID(c)) into EDB
    convert(c)
  end if
end for
if n is root then
  Dump all bindings as predicates into EDB
end if
```

An EDB converted from an AST contains almost all of the syntax information about the underlying Java program, and thus the EDB can be big even for a small program. For example, for a simple Java test program with about 100 lines of code, the generated EDB has more than 150 different kinds of predicates, and the number of all of the EDB predicates reaches more than 2,200. For a reference to all the different predicates that can be generated, interested readers can find them in Section 8.3 in the Appendix.

We carried out two case studies to use this AST-based model to implement two of the design rule checkers that we have also im-

plemented using our approach, and compared the implementations respectively.

5.2.1 Case study: Consider passing base types as parameters

The first case study is to experiment with implementing the FxCop design rule concerning with the reusability of an API method that we have discussed in Section 5.1.1.1. To facilitate the comparison between the two implementations, we intentionally use the same algorithm as we have used earlier.

Recall that the core of the algorithm was to develop a predicate called `TypeCantBeMoreAbstract` that tries to find a method call inside a method body implementation such that the invoked method is not declared by any super type of a parameter type. Very soon we hit the first difficulty when implementing this predicate using the AST model, which did not exist when using our shadow based approach. The problem is how we can find a method call invoked inside a method body. Sure, each AST edge predicate has the information about which node is the parent node and which are children nodes. But using this information to find a method invoked by a method body means we have to exhaustively iterate all possible edge paths from a `MethodDeclaration` node to a `MethodInvocation` node, which is impossible to do given we have more than 150 different kinds of predicates and the depth of the AST is unbounded.

To overcome this, we had to go back to the AST to EDB conversion algorithm to generate a new predicate called `HasChild(parent:ID, child:ID)` that records all of the parent/child relationships. The reason this will help is that now we can write a new predicate as in Listing 5.5 to compute the transitive closure of it and thus can find all

method calls beneath.

Listing 5.5: Compute transitive closure of parent/child relationship

```
CanReach(parent,child) :- HasChild(parent, child) .
CanReach(parent,child) :- HasChild(parent, middle),
                          CanReach(middle,child) .
```

It is worthwhile pointing out that even this is not exactly right. In this particular case, `CanReach` will find more method invocations than necessary since it follows every edge, e.g., it will mistakenly pick up a method call invoked inside an inner class defined within the source method implementation body, and this is wrong. In this case study, we use `CanReach` as an estimate of the precise result and ignore such kinds of edge cases.

After we get this issue resolved, implementing this design rule checker amounts to figuring out the right predicates to use among the over 150 different kinds of predicates. Our experience suggests this is much more challenging than using our shadow model based approach. Listing 5.6 is the code that implements this checker.

Listing 5.6: Passing base types as parameters checker in AST model

```
MethodParamTypeAt(m,p,t) :- MethodDeclaration_PARAMETERS(m,p,arg,_),
                            SingleVariableDeclaration(arg,m,_,_,_),
                            SVD_TypeBinding(arg,t) .
SVD_TypeBinding(arg,t) :- SingleVariableDeclaration_TYPE(arg,at),
                          SimpleType(at,_,t) .
MethodParameterTypes(m,t) :- MethodParamTypeAt(m,_,t) .
TypeCantBeMoreAbstract(m,t) :- MethodDeclaration(m,_,_,_,_),
                               CanReach(m,call), MethodInvocation(call,_,_,b),
                               MethodBinding(b,_,_,_,_,tb,_,_,_,_,_),
                               MethodDeclaration(callee,_,b,_,_),
                               TypeDeclaration(t,_,tb,_),
                               !SuperTypeHasDeclaration(t,callee) .
TypeCantBeMoreAbstract(m,t) :- MethodDeclaration(m,_,_,_,_),
                               CanReach(m,v),
```

```
                        VariableBinding(v,_,_,1,_,_,_,_,_,t,_) .
SuperTypeHasDeclaration(t,m) :- TypeDeclaration(t,_,tb,_),
                        SuperType(tb,tsupb),
                        TypeDeclaration(t2,_,tsupb,_),
                        CanReach(t2,m2), MethodDeclaration(m2,_,_,_,_),
                        MethodsMatchSig(m,m2) .

MethodsMatchSig(m1,m2) :- MethodDeclaration(m1,_,b1,0,_),
                        MethodDeclaration(m2,_,b2,0,_),
                        MethodBinding(b1,n,_,_,_,_,_,_,_,_,_,_),
                        MethodBinding(b2,n,_,_,_,_,_,_,_,_,_,_),
                        !NotMatchSig(m1,m2), !EQID(m1,m2) .
NotMatchSig(m1,m2) :- MethodDeclaration_PARAMETERS(m1,_,_,n1),
                      MethodDeclaration_PARAMETERS(m2,_,_,n2),
                      !EQNUM(n1,n2) .
NotMatchSig(m1,m2) :- MethodParamTypeAt(m1,p,t1),
                      MethodParamTypeAt(m2,p,t2),
                      !EQID(t1,t2) .
MethodShouldUseBase(m) :- MethodParameterTypes(m,t),
                    !TypeCantBeMoreAbstract(m,t),ClassOrInterface(t) .
```

The implementation basically keeps the same algorithm structure as we have used in our shadow based implementation. But we think readers will agree that this implementation is much harder to understand than our earlier solution, as there are much detailed predicates that need to be grasped. We just point out a few important points to help readers digest this.

The implementation relies heavily on several binding predicates to retrieve important type related information. For example, `MethodBinding` predicates tell us which method an invocation is calling and which method declaration has declared it; the `VariableBinding` predicate tells us if a variable access is actually accessing a field, and if so, which class has declared it.

When comparing this implementation with our earlier shadow

based implementation, besides this one is harder to understand, it is also obvious that it is much longer. This implementation has used 37 Datalog literals on the right hand sides of the inference symbols, while our earlier solution only used 20 Datalog literals. Also when writing this implementation, it was annoying to us that we had to deal with a lot of syntax details that are not directly relevant to the checking logic. For example, to define the predicate at line 1, we had to figure out we needed to go through the `SingleVariableDeclaration` node to get the type of one method parameter.

5.2.2 Case study: Law of Demeter checker

The second case usability comparison is done through implementing the static Law of Demeter (LoD) checker using the AST based model and compare it with the implementation using our proposed solution presented earlier. Again, to facilitate the comparison between the two implementations, we intentionally use the same algorithm as we have used earlier.

To make it convenient for readers to understand the code while connecting it with the LoD, we incorporate the definition of the static form of LoD here.

The static form of the LoD states that in the interest of decreasing coupling between classes, in the implementation of a method of a class, one can only call methods on the following set of types:

- the class itself;
- the types of the class's fields;
- the types of the method's formal parameters;
- the classes whose constructors are invoked in the method implementation;

- the return types of the methods called by the method.

Listing 5.7 is our Datalog implementation of the LoD checker basing on the AST model.

Listing 5.7: LoD checker in AST model

```
LoDViolate(c) :- MethodInvocation(c,_,_), !LoDConform(c) .
TargetType(target,t) :- SimpleName(target,t,_,_) .
TargetType(target,t) :- ThisExpression(target,t) .
CallWithTargetType(c,t) :- MethodInvocation(c,_,_),
                  MethodInvocation_EXPRESSION(c,target),
                  TargetType(target,t) .
CallWithinClass(c,t) :- MethodInvocation(c,_,_), CanReach(td,c),
                     TypeDeclaration(td,t,_) .
LoDConform(c) :- CallWithTargetType(c,t), CallWithinClass(c,t) .
ClassHasFieldType(t,ft) :- TypeDeclaration(td,t,_),
                        TypeDeclaration_BODY_DECLARATIONS(td,_,fd,_),
                        FieldDeclaration(fd),SimpleType(st,ft),
                        FieldDeclaration_TYPE(fd,st).
LoDConform(c) :- CallWithTargetType(c,t), CallWithinClass(c,ct),
                ClassHasFieldType(ct,t) .
CallWithinMethod(c,md) :- MethodInvocation(c,_,_), CanReach(md,c),
                MethodDeclaration(md,_,_,_) .
ContextHasArgType(md,at) :- MethodDeclaration(md,_,_,_),
                MethodDeclaration_PARAMETERS(md,_,arg,_),
                SingleVariableDeclaration(arg,_,_,_),
                SingleVariableDeclaration_TYPE(arg,st),
                SimpleType(st,at) .
LoDConform(c) :- CallWithinMethod(c,md), CallWithTargetType(c,t),
                ContextHasArgType(md,t).
ContextUsesCType(md,ct) :- MethodDeclaration(md,_,_,_),
                CanReach(md,constr),
                ClassInstanceCreation(constr,_,mb),SimpleType(st,ct),
                MethodBinding(mb,_,_,1,_,_,_,_,_,_,_),
                ClassInstanceCreation_TYPE(constr,st).
LoDConform(c) :- CallWithinMethod(c,md),CallWithTargetType(c,t),
                ContextUsesCType(md,t) .
ContextUsesReturnType(md,rt) :- CallWithinMethod(c,md),
                MethodInvocation(c,rt,_) .
LoDConform(c) :- CallWithinMethod(c,md),CallWithTargetType(c,t),
                ContextUsesReturnType(md,t) .
```

In the code, the five `LoDConform` predicate definitions appearing at line 9, 14, 23, 29, and 33 correspond to the five cases of the allowed LoD target types respectively.

The code uses 43 Datalog literals on the right hand sides of the inference symbols, and our implementation based on shadow model uses 29 literals. More importantly, as it is also evident from the previous case study, the AST based approach forces the user to take care of tiny syntactical details, which makes it harder to program and error prone. As such a concrete example, in our first version of this implementation, we did not have the second `TargetType` predicate at line 3. The checker failed to mark a method call on `this` type as an allowed call, even though we had the case be allowed as encoded in the `LodConform` predicate at line 9. Only after a careful study of the tree structure again, did we find out that an access to the `this` variable has a different expression node representation than a regular variable access. The latter is being presented as a node with label `SimpleName`, and the former is being presented as a node with label `ThisExpression`. It was necessary to add the second `TargetType` rule to accommodate this scenario.

An important implication of having to take care of syntactical details is that when we want to make queries to access information by doing joins on a few predicates, the exact traversal paths have to be right. When we programmed under this model, we had to frequently check the AST tree structure to make sure everything is right.

5.2.3 Conclusions from the experiment

Through the usability experiment, we conclude that the implementation by using our proposed approach is superior to the implementa-

tion by using the AST based approach in the following aspects:

- Our implementations tend to be more succinct and easier to understand. In particular, in the two case studies, using our approach, the implementations of the checkers respectively use 66% and 38% less Datalog literals than the checkers in the AST based approach.
- Our approach lets users focus on algorithm development other than on syntactical details.
- It is a challenging task to figure out which predicates one should use among more than 150 different AST-based predicate kinds.
- It is a challenging task to write queries against complicated AST structures.

We have to point out, however, that since the AST model provides richer information than what the shadow model presents, it can enable more design rule checker implementations than otherwise possible. For example, the *some methods should only be invoked inside **d**oPrivileged block* rule checker that is in the FindBugs evaluation study could not be done by using our approach, but one should be able to implement such a checker using the AST based approach. But those object/flow sensitive analysis still could not be done by simply querying the AST based model, which has the same limitation as our approach on this aspect. The user will have to switch to more semantic based data/control flow analysis tools to achieve this kinds of tasks.

5.3 Performance evaluation

Performance/scalability was one of the main design considerations when we started to build the system. It is important to evaluate the

performance of the system and how well it can scale to large real world program sizes.

5.3.1 Performance of analyzing Java benchmarks

We measure the performance of the system by running a few queries on the following 8 benchmarks. The benchmarks are real world Java applications in the form of JAR files that we downloaded from *SourceForge.net* or from *Eclipse.org*. Table 5.4 lists the 8 benchmarks and their sizes, ordered by the lines of code (LOC). The number of methods is the number of the method definitions (with method bodies) that a benchmark provides. The sizes vary from 929 methods to 18,221 methods, and in terms of the lines of code, they vary from 10KLOC to 505KLOC.

Benchmark	Abstract	Number of methods	Lines of Code (KLOC)
JUnit	Unit testing framework	928	10
PdfSam	PDF split/merge tool	1012	45
iText	Java PDF library	4603	136
FindBugs	Bug finding tool	9077	186
jFreeChart	Java chart library	7045	217
Saxon-HE	XSLT/XQuery Processor	10268	265
JDT-core	JDT core library	16302	435
Weka	A machine learning tool	18233	505

Table 5.4: Summary of the benchmarks

Table 5.5 lists the queries that we run against the benchmarks and their abstracts. Each query is given a short-name code so that we can refer them later in tables.

Query (code)	Abstract
CheckHashEq (CH)	Query to see if every equals method has an accompanying hashCode method
CheckFields (CF)	Query to see if every couple of equals/hashCode uses the same set of fields
HideBase (HB)	Query to see if a subclass's method will hide a superclass' method
PassBase (PB)	Query to see if a method parameter's type can be more abstract
LoD (LoD)	The Law of Demeter query
AspectJQ (AQ)	A query equivalent of AspectJ pointcut *execution(* Collection.add*(..))*

Table 5.5: Summary of the queries

Our tool is integrated with the whole AspectJ compiler compilation and weaving process, but in this evaluation, we are primarily interested in the running time of the Datalog solving part itself. So in the experiment, we isolate the Datalog solving part from the rest of the compilation process, and the running time presented here is referring to the time for the Datalog solver to solve the generated shadow database and to compute the results. The benchmarks were run on an Intel Pentium-4 3.0GHz single processor machine running the Linux operating system and the JRE 1.6 with 1GB of physical memory. Unless otherwise specified, the JVM is started with 1GB memory as the maximum allowed heap space.

Table 5.6 lists the running time of each query on every benchmark. Due to the space constraint, the queries are identified using the codes listed in Table 5.5. Figure 5.1 is the chart to visualize the same data. From those experiments, we conclude that our approach can indeed scale to large real world applications. Earlier similar work [5] only established running time on benchmarks up to 100K lines of code, on less diversified queries.

Benchmark	Size (KLOC)	Running Time for Query (s)					
		CH	CF	HB*	PB*	LoD*	AQ*
JUnit	10	1.4	1.6	5.7	2.9	2.9	1.5
PdfSam	45	1.4	2.5	8.6	6.7	5.4	2.6
iText	136	3.6	10.3	28.9	14.6	18.1	6.9
FindBugs	186	7.0	14.2	100.2	34.9	37.4	20.2
JFreeChart	217	7.9	11.1	48.1	20.3	21.3	10.3
Saxon-HE	265	8.0	15.9	56.4	26.0	33.0	13.4
JDT core	435	12.0	45.2	177.1	59.5	95.4	31.1
Weka	505	13.8	38.8	172.0	74.4	346.4	37.2

Table 5.6: Running time of queries on benchmarks

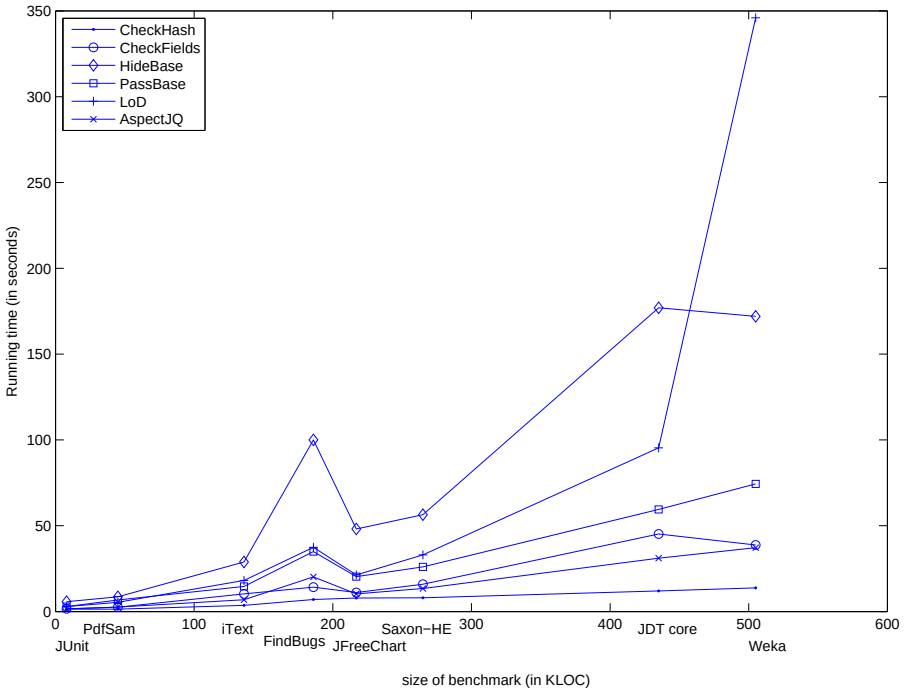

Figure 5.1: Chart of running time of queries on benchmarks

In Table 5.6, there are four queries that are marked with *, which indicates that we have used explicit BDD variable orders in the experiments, instead of using the default one determined by the *bddbddb* solver. As we know that BDD variable ordering has significant impacts on the size of the result BDD diagrams, yet determining the

optimal order is an NP-complete problem [9] unfortunately. Several heuristics about variable ordering have been suggested to help speed up the solving [1, 12]. We found one heuristic to be particularly effective in our experiments, i.e., when there are equality relations on fields, their variable encoding should be interleaved. This applies to many join operations that we have in those four queries.

5.3.1.1 Performance impact of BDD variable ordering

As having been mentioned earlier, we have used explicit BDD variable orderings in 4 queries in this performance study. In all of the 4 cases, the variable orderings involve interleaving the bits of the variables between which there is an equality relationship, such as there is a relational join operation on them. The performance impact of using this particular ordering technique is significant, as evident from the following table. Table 5.7 summarizes the query speed up ratios for the 4 queries on the 8 benchmarks after using explicit BDD variable orderings. For each query, let the query time when using explicit ordering be T_O, and the query time without using explicit ordering be T_{NO}, the query speed up ratio is $\frac{1/T_O - 1/T_{NO}}{1/T_{NO}} = \frac{T_{NO} - T_O}{T_O}$.

Benchmark	Size (KLOC)	Speed up ratio			
		HB	PB	LoD	AQ
JUnit	10	1.8%	186.2%	82.8%	146.7%
PdfSam	45	15.1%	177.6%	107.4%	207.7%
iText	136	144.6%	559.6%	242.5%	558.0%
FindBugs	186	202.3%	1422.3%	767.9%	910.9%
JFreeChart	217	205.2%	931.0%	331.5%	640.8%
Saxon-HE	265	445.0%	1346.9%	513.3%	952.2%
JDT core	435	338.9%	1858.5%	412.6%	1463.0%
Weka	505	436.3%	1517.5%	178.9%	1623.7%

Table 5.7: Query speed up ratio when using explicit BDD variable ordering

As can be seen from the table, after using the explicit variable ordering technique, the queries typically run several times faster than using the default orderings determined by the solver. In the best case of query PB on benchmark JDT core, the explicit ordering delivers a speed up ratio of more than 18 times!

5.3.1.2 Performance impact of memory made available

In Table 5.6, the running time have been collected with the Datalog solver being allowed to use up to 1GB memory, as this is the memory size suggested by the solver (*bddbddb*). It is interesting to see how a query time changes, if there is any change, when the Datalog solver is given different memory upper bounds. To see it, we have chosen one query, i.e., query CF, and collect the running time of the query on the 8 benchmarks above, under the conditions of the Datalog solver being given different memory limits. In this study, the given memory limits are 64MB, 128MB, 256MB, 512MB, and 1GB respectively. Table 5.8 summarizes the running time of query CF on the benchmarks under different memory conditions. In the table, OOM stands for the query for the given benchmark running out of the memory limit.

Benchmark	Size (KLOC)	Running Time for Query CF with various memory limits (s)				
		64MB	128MB	256MB	512MB	1GB
JUnit	10	1.7	1.7	1.7	1.7	1.7
PdfSam	45	2.8	2.8	2.8	2.8	2.8
iText	136	OOM	10.3	10.4	10.3	10.4
FindBugs	186	OOM	20.7	14.7	14.7	14.6
JFreeChart	217	OOM	11.7	11.1	11.1	11.1
Saxon-HE	265	OOM	28.0	16.6	16.6	16.6
JDT core	435	OOM	OOM	OOM	42.1	40.0
Weka	505	OOM	OOM	102.0	35.9	35.8

Table 5.8: Running time of query CF on benchmarks with different memory limits

The impacts of the memory made available on the query time are twofold. First of all, the Datalog solver could run out of memory when the given memory limit is insufficient. Second, increase of memory limit can improve the query time, but the improvement typically is not very significant. In Table 5.8, except those OOM cases, there are only three cases where the increase of memory limit notably decreases the query time (with the query time difference being bigger than 10%). In those three cases, from the debugging information available from the Datalog solver, it appears that more time are spent on garbage collection activities when the memory limit is lower.

5.3.2 Performance of running call graph analysis

There is one kind of queries that are particularly expensive to run, that is the queries that involve call graph reachability analysis that we have presented earlier. We have shown that our approach could be used to find subtle infinite call chain bug introduced by careless aspects. So in this experiment, we evaluate the scalability of this kind of queries separately.

The methodology used in this study is as follows. We first weave

a simplified version of our AspectJ based Law of Demeter dynamic checker, which we have presented earlier in Listing 4.15, into the benchmarks used in the previous sub-section, so that we can apply our infinite call chain detection algorithm on programs advised by the aspect. Then we run our shadow model based infinite call chain static detector on the woven applications plus the accessed Java system classes. The running time of the infinite call chain detection algorithm is collected.

Among the 8 benchmarks presented earlier, we were only able to weave the LoD dynamic checker aspect into 5 of them, which are `JUnit`, `PdfSam`, `iText`, `JFreeChart`, and `Weka`. We could not weave the aspect into benchmark `FindBugs` because it refers to classes provided by Apple's Java Extensions library that we do not have access to. When weaving an aspect into a base program, the AspectJ compiler requires all classes referred by the base program be accessible in the class searching path. We could not weave the aspect into benchmark `Saxon-HE` and benchmark `JDT-core` because in both cases, the weaving (using Eclipse AspectJ version 1.5) stopped prematurely, citing it has hit the code size limit when generating methods, which is due to a technical limitation of AspectJ's bytecode weaving approach. The woven version of benchmark `Weka` is so big that it won't terminate just to use our tool to count the number of method calls in it, letting alone doing any other analysis. So in this study, we only use the woven versions of the rest 4 benchmarks to carry out the evaluation.

5.3.2.1 Code size impact of the weaving

The LoD dynamic checker aspect makes invasive code changes to the base programs, as it advises every method execution and every method call. For each shadow that it advises, the compiler generates

a few utility method calls for housekeeping purposes, such as creating runtime join point objects, creating join point signature objects, and determining if an aspect instance is present for a given target or hosting object, etc. So in the woven versions of the 4 benchmarks, the numbers of method call-sites are a few times more than in the original benchmarks. The number of method definitions has little change before and after weaving for each benchmark, with only 3 methods being added to each benchmark after weaving. Table 5.9 summarizes the number of method calls before and after weaving respectively for the 4 benchmarks, and the number of method call increase ratios for them. The lines of code for the woven versions are not available as the weaving is done at the byte code level.

Benchmark	Num of method calls(pre-weaving)	Num of method calls(post-weaving)	Num of method calls increases by
JUnit	2573	23450	811.4%
PdfSam	6014	45224	652.1%
iText	23928	177946	643.7%
JFreeChart	26917	230362	755.8%

Table 5.9: Impact of the weaving on number of method calls

5.3.2.2 Call graph analysis performance

Then we run our shadow model based infinite call chain static detector on the woven applications plus part of the Java run time library classes, which include the classes within the package of `java.lang` and the accessed classes within the package of `java.util`, so that we have the whole call graphs.

Table 5.10 lists the sizes of the four analyzed benchmarks in terms of the number of method definitions and the number of method calls, and the running time of each benchmark by applying the infinite call chain detection algorithm on it. Note that for each benchmark,

the number of method calls includes the corresponding number of method calls in Table 5.9 and the number of method calls in the aforementioned Java system library classes that are analyzed by the checker altogether. The meaning of the number of method definitions is similar. The lines of code measurement for each benchmark is not available as the benchmarks have been advised by the dynamic LoD checker aspect at the byte code level.

Benchmark (woven+system classes)	Num of method definitions	Num of method calls	Running time (s)
JUnit	1912	25914	64.4
PdfSam	1996	47688	76.1
iText	5587	180410	1172.2
JFreeChart	8029	232826	$> 4h$

Table 5.10: Benchmarks for infinite call chain analysis and running time

It is clear from the table that the call graph reachability analysis takes much longer to run than other analysis that we have experimented with earlier. But the table also shows that it is feasible to do so for non-trivial size of applications.

We did not use explicit BDD variable ordering when collecting the running time for this experiment. One might wonder if explicit BDD variable ordering might help in this case. Unfortunately, we have tried the heuristic that worked well in the previous experiment, not only it could not help in this experiment, but it made it worse. For example, the checking of the second benchmark was not able to finish when using the explicit ordering, as it ran out of memory. This suggests that a heuristic working well for one query does not necessarily work well for another.

5.3.3 Performance of earlier analysis tasks on woven Java benchmarks

We are interested in knowing how bad an aspect can affect the performance of the analysis tasks that we have done in the first sub-section of this performance evaluation. With those woven versions of benchmarks presented in the previous sub-section, we have an opportunity to carry out such an evaluation. We have applied those 6 analyses presented in the first performance evaluation experiment on the 4 woven versions of benchmarks, and we have collected the running time of each analysis. The result is presented in Table 5.11.

In the table, again, the four queries that are marked with * are queries for which we have used explicit BDD variable orders, instead of using the default one determined by the *bddbddb* solver. The lines of code measurements are not available since the benchmarks are woven versions, but the number of method calls in each benchmark is available from Table 5.9.

Benchmark (woven)	Running Time for Query (s)					
	CH	CF	HB*	PB*	LoD*	AQ*
JUnit	1.4	1.7	6	7.4	9.5	5.6
PdfSam	1.5	2.8	8.9	16.3	17.8	9.9
iText	4.0	10.6	32.8	71.7	327.2	43.5
JFreeChart	10.5	11.8	52.8	121.5	$> 4h$	77.9

Table 5.11: Running time of queries on woven versions benchmarks

To evaluate the impact of the dynamic LoD checker aspect weaving on the query time, we compare the running time of each query in Table 5.11 with the running time of the same query on the corresponding pre-woven versions of the benchmarks listed in Table 5.6. The number of method call difference between the respective pre- and post-woven versions of the benchmarks has been presented in Table 5.9, and there is little change in the numbers of method defini-

tions before and after weaving (each woven version only gets 3 more method definitions than before weaving). Table 5.12 summarizes the ratios of running time increase for each case.

Benchmark (woven)	Increase ratio					
	CH	CF	HB*	PB*	LoD*	AQ*
JUnit	0.0%	10.0%	5.3%	155.2%	227.6%	273.3%
PdfSam	7.1%	12.0%	3.5%	143.3%	229.6%	280.8%
iText	11.1%	2.9%	13.5%	391.1%	1706.7%	530.4%
JFreeChart	32.9%	6.3%	9.8%	498.5%	> 67500.0%	656.3%

Table 5.12: Ratios of query time increase on woven versions

From the table, we can tell that the impact of an aspect on an analysis depends on the nature of the aspect and the analysis. In this particular study, the dynamic LoD checker aspect results in a big change on the number of method call-sites in the woven program, and so it has a dramatic impact on the analyses that are sensitive to method calls. This is evident from the running time increases of queries `PB`, `LoD`, and `AQ`. In those three queries, the running time increases by 143.3% at the lower end and by 1706.7% at the higher end. In the extreme case of the LoD query, benchmark `JFreeChart` even fails to terminate after 4 hours of running while the same query on the pre-woven version of the benchmark could finish in tens of seconds. On the other hand, the aspect has little impact on the analysis time of the other three queries that are not sensitive to method calls, which are queries `CH`, `CF`, and `HB`. They are sensitive to the number of method definitions and/or the number of field access shadows, instead. Most of the running time for those three queries on the benchmarks remain very close to the previous numbers. Even in the worst case of query `CH` on benchmark `JFreeChart`, the running time only increases by 32.9%.

5.3.4 Conclusions from the experiment

This performance evaluation experiment suggests that our approach and the system indeed can scale well to large size real world programs on non-trivial queries.

In particular, we have established the query running time on benchmarks of up to 500KLOC, while the earlier similar work [5] only established running time on benchmarks of up to 100K lines of code. In addition, we have evaluated the performance on more diversified queries, including a call graph analysis query, while the queries experimented by the earlier similar work [5] were all focused on the queries that are expressible in the AspectJ native pointcut language, like query `AQ` presented in this study. For the call graph analysis, which is more expensive in general, we show that our system can scale to reasonably big programs, such as a program with 180K method call-sites. The impact of the aspect weaving on analyses largely depends on the nature of the aspects and the analyses.

CHAPTER 6

Related Work

In this chapter, we go over the existing research work that is related to this dissertation work.

6.1 Datalog as Pointcut Designator Language

To our knowledge, the work [5] by Avgustinov *et. al.* is the only other work that suggests to use Datalog as a pointcut language, and they were the first to make such a proposal. In [5], they show that Datalog is a suitable intermediate language to interpret the semantics of the existing AspectJ's native static pointcut designators by translating the latter to appropriate Datalog queries. Their design goal is different from ours in that we wanted to leverage Datalog's query power to extend AspectJ's pointcut specification and evaluation mechanism for the purpose of capturing various software design rule constraints. At the design and implementation level, the differences between the two systems include: (1) our Datalog based system is integrated with the existing AspectJ pointcut evaluation mechanism so that we can bring in the solved Datalog shadow results and use them to refine the results of the native pointcut expressions, while their system lacks this

capability; (2) the concrete designs of the Datalog shadow representation are different as a result of our intention to leverage the reduction power of the BDD operations; (3) at the implementation level, their system relies on a relational database to solve Datalog queries, while our system uses an advanced BDD-based Datalog solver to do so. Their work demonstrates it is feasible to write Datalog queries to interpret native AspectJ pointcuts for programs of up to 100K LOC. Our evaluation shows that our system can scale well to programs of up to 505KLOC for customized queries (typically in-expressible in and more complex than AspectJ native pointcuts).

6.2 Static Aspect Languages for software style rule checking

There are at least two static aspect language work [4, 49] aiming at enforcing software style or design rules, directly based on AOP's language model, inspired by our earlier work of [34] and [66].

In [4], Aotani and Masuhara present an AOP compiler called *SCoPE*. It extends AspectJ language's conditional pointcut, i.e., the `if` pointcut, by allowing the expressions within the `if` pointcut to have access to the compile time reflective information to carry the computation. Then the compiler does an analysis trying to determine whether only statically evaluable expressions are being used within an `if` pointcut and if it is so, the compiler directly evaluates the `if` pointcut statically, instead of emitting a runtime test code, as a regular AspectJ compiler would do. It has been shown that this way, a few program style rules can be statically enforced. Their work is very similar to our Statically Executable Advice (SEA) approach in that both are built up on the existing AspectJ language's static shadow model and are extensions to the compiler without drastic changes to the language itself. Our

SEA approach differs from theirs in that our approach allows static checking logic to be defined on ordered compile time shadow events that is more aligned with the commonly used Abstract Syntax Tree traversal approach. Their approach, on the contrary, supports static checking logic defined on each isolated static join point, which makes it difficult to encode those design rules that are concerned with relationships between static shadows. As an example, in their implementation of the static checker for the LoD (as Figure 4 in their paper), they omit the sub-rule related to constructors, and it is not clear how an implementation in their approach would look like if that sub-rule were to be enforced. On top of that, our Datalog based approach has additional advantages over their approach (and our SEA approach too) primarily thanks to the declarativeness of the query language.

Morgan, De Volder, and Wohlstadter [49] present a domain specific static aspect language called *Program Description Logic (PDL)* for implementing customized design rule checkers, also within the framework of the AspectJ language. Realizing AspectJ's existing pointcut designators are not sufficient for capturing many important design rules, they introduce a set of new pointcut primitives that are specially designed for encoding design rules. We believe our approach has a better integration with the current AOP languages because it is directly based on the existing shadow model and only extends the query mechanism. In addition, similar to the problem associated with Aotani and Masuhara's approach, it is hard, if not impossible at all, to use their approach to encode design rules that are concerned with relationships between static shadows, such as the LoD design rule and the `hashCode/equals` rule presented earlier.

6.3 Generic Code Query Systems

This dissertation work is also related to generic code query systems, which could also be used to detect software design rule violations, among other things. There has been a long history of research effort in this area, and it is not possible to cover all of them here. So we only review the most relevant ones published in recent years. The difference between our approach/system and all other systems is characterized by our thesis statement: our system operates on the existing AspectJ's shadow model that is well defined and is familiar by AOP programmers, and our system has a tight integration with the AspectJ language and the compiler.

Crew has designed and implemented a C++ source code query system called ASTLog [17]. It supports queries in Prolog-like logic programming languages over an abstract syntax tree constructed by compiling C++ source code. It has been reported that queries written in this system can scale reasonably well to real world program sizes. The system is operating on the abstract syntax tree model, whose detail is not completely revealed. But according to our experience from the usability study in the evaluation chapter, it could be painful to implement many software design rule checkers in this approach, as one has to deal with much low level syntactical details present in an AST.

Jazen, McCormick, and Volder [31, 48] have designed the JQuery system, a Java source code query plugin for Eclipse. A nice feature about their system is that they allow the user to customize the Eclipse code navigation operations by associating a menu item with a rule written in a logic programming language named TyRuBa, which essentially is a query over a code database. The queries are operating on a data model that they specially designed for the system, and the

data are generated dynamically by calling Eclipse APIs to retrieve from the IDE.

In [23], Hajiyev, Verbaere, and Moor presented a Java source code query system called CodeQuest, which uses Datalog as the query language due to the need to write recursive queries and the desire to have the guarantee of termination. The program database to be queried against is stored with commercial relational DBMS systems and a Datalog query is translated into the corresponding SQL statements with some special optimizations. The details of the data model on which those queries operate on, however, are not published.

The Java Tools Language (JTL) [15] is yet another Java code query system. It features a specially designed surface language, JTL, similar to the Java language syntax itself, with Datalog under the hood serving as the query engine. This surface language is one of the most visible strengths of the system, as the queries written in this language just look like Java code, and this would be very attractive to Java programmers that do not want deal with Datalog directly. On the other hand, this surface language also makes it cumbersome to express some operations that are natural to do in Datalog, such as the join operation. Another limitation of their system is that it lacks information within a method body, and thus any query that needs this information, e.g., which methods have been called by method `foo`, cannot be supported.

More recently, Moor *et. al.* have presented a new code query system, based on a domain specific language named .QL [19] that they have designed. It is a SQL like language, but with some Object-oriented features, including methods, inheritance and dynamic binding. The OO features are for the purpose of reusing queries that have already been written. The language is actually a surface language with the queries delegated to Datalog behind the scene, to narrow

the gap between Datalog and the user community. They have also developed a proprietary Datalog optimization technique [18] based on type inference and type erasure and specialization. We believe that as a future work, our system can also benefit from this particular optimization. We discuss some of the details in the future work section of the concluding chapter.

6.3.1 CTL for control flow path query

Work on program control flow path query is relatively rare compared to the work on program structural queries. Recently researchers have proposed to use variants of Computational Tree Logic (CTL) to implement control flow path queries [10]. CTL [14] is a well known temporal logic specification language designed for model checking, and it has been used to model check properties ranging from hardware verification to program analysis.

In [10], Brunel *et. al.* proposed an extension of CTL, called CTL-VW, as a pattern matching language for control flow paths to facilitate program transformations, based on earlier work of CTL-FV and CTL-V [38, 37]. CTL-FV and CTL-V allow CTL formulas to have free variables and existentially quantified variables respectively, so that they have a richer semantics for control flow path matching. On top of that, CTL-VW allows the bindings of the existentially quantified variables to be preserved, instead of being discarded after matching, so that the bindings can be used by a transformer to refer to program elements mapped to by meta-variables in the bindings. Typically, such a transformer needs to modify a program's control flow logic when a pattern specified in CTL-VW is matched.

Similar to our work, their work is also about reasoning about program structures, in particular, control flow path structures in their

case. There is a particular relevance, since people have shown CTL can be expressed by a proper subset of stratified Datalog, called Datalog-LITE [21].

6.4 Datalog and BDD for Program Analysis Tasks

Datalog was originally invented by the Database community [63]. While it has not been very successful among commercial database systems, it has found many applications outside of the database area, primarily for program analysis tasks.

Researchers have been using Datalog as a query language to solve program analysis problems for long time. Some examples are presented in [47, 63, 55, 7]. More recently, it has drawn more research attention, partly because of the work of Whaley *et.al.* [65, 39, 1].

In [65], Whaley and Lam have shown Datalog can be used to model the context-sensitive pointer alias analysis problem, and more importantly, by leveraging the optimization techniques offered by BDD, they show the approach can scale to large size real world programs. Context-sensitive pointer alias analysis is generally considered a hard problem in the program analysis community, due to the huge number of contexts that need to be handled. In their benchmarks, they show their approach can scale to programs with up to 10^{14} contexts. As a result of this work, they have also delivered a practically useful generic BDD based Datalog solver named *bddbddb* [1], on which the main part of this dissertation work is based.

In [39], Whaley *et.al.* show the same technique can be used to help discover security issues in Database applications [39].

In the previous section, we have found that a few code query systems are also using Datalog as the query language.

There is also a progress in the complexity analysis of Datalog evaluation recently. In [47], McAllester has shown that while the commonly recognized datalog program running time complexity of $O(n^k)$, where k is the largest number of variables in any single rule, and n is the size of the EDB, is true, the real complexity could be more refined and more efficient. The key observation is that variable counting, which is how the $O(n^k)$ complexity is determined, is too crude for many algorithms. A new notion of *prefix firing* is defined and it is shown that the running time complexity is bound by the size of EDB plus the number of prefix firings that is more refined than simply $O(n^k)$. The paper also argues that bottom-up logic program presentations of static analysis algorithms are clearer and simpler to analyze, by doing several case studies of static analysis algorithms. This finding is also echoed by the popularity of Datalog in program analysis tasks and code query systems.

Outside of the program analysis area, Ou [52] has used Datalog to model the network security and access control problem and shows it is feasible and desirable to do this way. Even in an apparently unlikely area, Loo *et. al.* have used Datalog to model distributed network routing protocols [44] as an alternative to the traditional approaches and have shown the protocols expressed in their approach are compact, clean and easier to adopt optimizations.

Using BDD for program analysis problems has been rare, but there is a growing interests of doing this recently, mostly due to its capability to compactly represent large relation sets, and its direct support of many relational operations. Whaley *et. al.* have used BDD to speed up Datalog program evaluation to solve the context-sensitive pointer alias problem by leveraging BDD's excellent capability to handle with relations with much redundant information. To our knowledge, the idea of using BDD to speed up logic program evaluation can be traced

back to the work of Iwaihara and Inoue [30] in 1995.

Berndl *et. al.* [6] have reported positive experience of modeling program analysis problems like subset-based points-to analysis by using BDD operations through a Java wrapper. They show the approach can scale well to large programs thanks to BDD's effectiveness of handling large relation sets. Realizing directly operating on BDD diagrams is too low level, and thus is difficult for understanding and is error prone, Lhoták and Hendren developed a higher level domain specific language called Jedd [40] to abstract BDDs as database style relations and to provide static type rules to make sure relational operations are used consistently with regard to the typing.

Zhang, Gupta and Zhang [67] store dynamic slices of program execution for debugging, which require large space, in BDDs. They show that after using the BDD representation, the space needed is greatly reduced.

CHAPTER 7

Concluding remarks and future work

7.1 Concluding remarks

This dissertation argues for using the shadow model of AspectJ-like Aspect-oriented programming (AOP) languages as the meta-model to build software static checkers to enforce software design rules to address the problems of the existing approaches. The existing approaches fall short due to one or more than one of the following reasons: (1) Specialized static checkers can only check a pre-defined rule set, lacking the capability for customization; (2) Specialized static checkers allow some customizations, but a customization is generally very difficult as the programmers have to use low level introspection APIs; (3) Generic code query systems tend to require the user to get familiar with the tool specific data models and/or the tool specific query languages.

To carry out the dissertation research, we started with implementing two AspectJ based dynamic checkers for two forms of the Law of Demeter, a software design rule. Through the experiment, we assess that the AspectJ language's two major components, i.e., the pointcut designator language and the shadow information made available to runtime advice, have already had sufficient support for one to imple-

ment a static checker for the statically checkable form of LoD, and what is missing is just a more expressive query mechanism.

To support our thesis, we go on presenting two shadow model based language extensions to AspectJ, i.e., a lightweight extension called *statically executable advice*, and a more ambitious Datalog-based pointcut evaluation system, with the focus on the latter system. Besides based on the shadow model that is familiar by AOP programmers, our Datalog-based system features a support of logic style declarative queries, a tight integration with the existing AspectJ native pointcut evaluation process, and a BDD based underlying shadow representation.

Finally, through the evaluation we show that our system can indeed be used to enforce a wide variety of real world software design rules, its usability is superior to a system built on an alternative meta-model, i.e., the Abstract Syntax Tree model, and it can scale well to real world program sizes.

7.1.1 Strengths and limitations of our system

The strengths of our system include:

- The underlying query data model is based on AspectJ's shadow model that average AOP programmers are already familiar with. This not only will make it easier for the programmers to write queries, but also will make it easier to port the system to Aspect-oriented variants of other languages, like C#,C++, etc., given the AspectJ programming model is widely adopted in the community.

- Using the shadow model as the query meta-model also helps implementing software design rule checkers as it allows higher

level abstraction than alternative models, such as the abstract syntax tree model.

- The system's tight integration with the AspectJ compiler framework makes it convenient to use our tool in their usual development workflow, as it is a natural enhancement to the existing AspectJ's `declare error` mechanism that was designed for simple static checking tasks.

- The support of the integration of logic style Datalog declarative queries and the AspectJ's native pointcut language make it possible for programmers to get benefits from both camps. They can use the Datalog approach to express more complex structural queries and use the native pointcuts to conveniently express syntax pattern based selection and then combine the results together.

- The special performance design choice for the system makes it scale well to real world program sizes.

The most visible limitation of our system is that its expressiveness is limited to design rule problems that the given shadow model can express. And extending the AspectJ's shadow model is a non-trivial task that involves a careful design and significant implementation effort even for a person familiar with the AspectJ compiler implementation. As we have shown in the evaluation, there are some real world design rules that do require information that is not present in the AspectJ's shadow model. Under such circumstances, the user would have to turn to other approaches or tools to enforce those design rules. Another limitation of our system is that for programmers that are not used to the Datalog syntax, they may find a specially designed surface language would be desirable. One possible solution is

to adopt a surface syntax of other systems like the .QL's [19], but still operating on the shadow meta-model.

7.2 Future work

We propose some future work to extend our system.

7.2.1 Conflict between efficiency and usability

Recall that earlier that we have specially designed the Datalog shadow representation in the interest of taking advantage of the reduction power of the BDD. One such an example is that we abstract out the method/constructor parameter type signatures into a separate domain, `PSig`. A design that is in favor of the performance is not necessarily in favor of the usability. Just as an example, in our system, one would have to write the following lengthy program in Listing 7.1, if she needs to write a query to retrieve all method calls that are within the shadow represented by the AspectJ native pointcut `execution( void A.compute(int,boolean))`, which captures any method definition provided by class `A`, whose name is `compute`, whose return type is `void`, which has two parameters with the type of `int` and `boolean` respectively.

Listing 7.1: A lengthy Datalog query

```
Declarations
  pointcut returnCalls(sh:SH) .
  methodDef(sh:SH) .
Definitions
  methodDef(sh) :- MethodExec(sh,sig),Signature(sig,"A","compute",_),
                   CodeSignature(sig,psig),
                   CodeSignatureNumParams(psig,2),
                   CodeSignatureParam(psig,0,intT),
                   TypeInfo(intT,"int"),
                   CodeSignatureParam(psig,1,boolT),
                   TypeInfo(boolT,"boolean"),
                   MethodSignature(sig,voidT), TypeInfo(voidT,"void") .
  returnCall(sh) :- MethodCall(sh,_,enclosing),methodDef(enclosing) .
```

In `methodDef`, it takes 10 predicates to capture what could be easily described in a short phrase in the native pointcut, with 6 of them dealing with the `PSig` domain. Note that our earlier claim that the integration of native pointcuts and the Datalog pointcuts would improve the usability does not help in this particular case, as the integration mechanism would only help when there is a logical interaction between a native pointcut expression and a Datalog pointcut expression. In this particular case, however, the `methodDef` predicate is just an interim predicate, only visible to another Datalog predicate.

This conflict between the efficiency and the usability could be much improved by introducing a lightweight domain specific language that can be embedded in a Datalog program, which can be done as a future work. In particular, for a query that is better done in native pointcuts, we could allow the user to use native pointcuts, and then bind the results to some Datalog "variable" that can be referred to by other Datalog pointcuts. For example, in our vision, with this new mechanism, the aforementioned query could be much simplified

into the following code in Listing 7.2:

Listing 7.2: A simplified Datalog query using native pointcut

```
Declarations
  pointcut returnCalls(sh:SH) .
Definitions
  returnCall(sh) :- MethodCall(sh,_,?enclosing) where
                  ?enclosing=execution(void A.compute(int,boolean)).
```

In a nutshell, we can introduce a `where` clause in a Datalog query, which can introduce a variable binding (with the `?` mark to flag it is a special binding) with a native pointcut. The special variable binding can be used anywhere that a regular Datalog variable can be used. Then in our system, we can introduce a preprocessing stage, where such an embedded native expression will be translated into the corresponding Datalog predicates like we have seen in Listing 7.1. This translation should not be hard to implement.

7.2.2 Leverage type based optimizations

Moor *et. al.* have developed some proprietary Datalog optimization techniques [18] based on type inference algorithms. We feel there are cases in programs written in our system that can benefit from their optimization techniques, especially the *type erasure* and *type specialization* optimizations. This should be an interesting and worthwhile future work.

Let's look at two code examples in Listing 7.3 and Listing 7.4 below, which respectively represents a case that can be optimized by using type erasure, and a case that can be optimized by using type specialization.

Listing 7.3: A type erasure example

```
Declarations
pointcut return(sh:SH) .
helper(sh:SH, sig:Sig) .
Definitions
helper(sh,sig) :- MethodCall(sh,sig,_),Signature(sig,t,"foo",_),
                  IsClass(t) .
return(sh) :- MethodCall(sh,_,_), helper(sh,_) .
```

The program in Listing 7.3 tries to find and return any method call shadow that is invoked on a method named `"foo"` and on a target type that is a class. The program is legal but it is clear that the second `MethodCall` predicate that is used for defining the `return` pointcut is redundant, as the `helper` predicate has already checked this, and there is only one disjunction case for the `helper` predicate. By using their type erasure optimization, we should be able to optimize out the second `MethodCall` predicate from the program, and thus boost the performance.

Listing 7.4: A type specialization example

```
Declarations
pointcut return(sh:SH) .
helper(sh:SH, sig:Sig) .
Definitions
helper(sh,sig) :- MethodCall(sh,sig,_),Signature(sig,t,"foo",_),
                  IsClass(t) .
helper(sh,sig) :- ConstructorCall(sh,sig,_),Signature(sig,t,_,_),
                  IsClass(t) .
return(sh) :- MethodCall(sh,_,_), helper(sh,_) .
```

On the other hand, the program in Listing 7.4 would return the same result as the previous program, however, the `helper` predicate has the second disjunction now, which requires a call be a constructor call made on a class. Certainly the second disjunction of predicate

`helper` is a useless computation, as the final `return` predicate requires the returned shadow be a method call shadow and thus will never satisfy it. Using their type specialization optimization, we should be able to optimize out the whole second disjunction of the `helper` predicate from the program, and thus boost the performance.

Bibliography

[1] bddbddb home page at sourceforge. http://bddbddb.sourceforge.net.

[2] The Byte Code Engineering Library. http://jakarta.apache.org/bcel/.

[3] The Motor Industry Software Reliability Association. http://www.misra.org.uk/. Continuously updated.

[4] Tomoyuki Aotani and Hidehiko Masuhara. Scope: an aspectj compiler for supporting user-defined analysis-based pointcuts. In *AOSD '07: Proceedings of the 6th international conference on Aspect-oriented software development*, pages 161–172, New York, NY, USA, 2007. ACM.

[5] Pavel Avgustinov, Elnar Hajiyev, Neil Ongkingco, Oege de Moor, Damien Sereni, Julian Tibble, and Mathieu Verbaere. Semantics of static pointcuts in AspectJ. In Matthias Felleisen, editor, *Principles of Programming Languages (POPL)*. ACM Press, 2007.

[6] Marc Berndl, Ondrej Lhoták, Feng Qian, Laurie Hendren, and Navindra Umanee. Points-to analysis using bdds. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 103–114, New York, NY, USA, 2003. ACM.

[7] F. Besson and T. Jensen. Modular class analysis with Datalog. In *SAS '03: Proceedings of the 10th International Symposium on Static Analysis*. Springer, 2003.

[8] Joshua Bloch. *Effective Java Programming Language Guide*. Sun Microsystems Inc., 2001. Pages 36-41.

[9] Bollig and Wegener. Improving the variable ordering of OBDDs is NP-complete. *IEEETC: IEEE Transactions on Computers*, 45, 1996.

[10] Julien Brunel, Damien Doligez, René Rydhof Hansen, Julia L. Lawall, and Gilles Muller. A foundation for flow-based program matching: using temporal logic and model checking. In *POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 114–126, New York, NY, USA, 2009. ACM.

[11] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, 35(8):677–691, 1986.

[12] Michael Carbin. Learning effective bdd variable orders for bdd-based program analysis. Master thesis at the Stanford University, May 2006.

[13] A. Chandra and D. Harel. Horn clauses and generalizations. In *Journal of Logic Programming*, volume 2(1), pages 1–15, 1985.

[14] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[15] Tal Cohen, Joseph (Yossi) Gil, and Itay Maman. Jtl: the java tools language. In *OOPSLA '06: Proceedings of the 21st annual*

ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, pages 89–108, New York, NY, USA, 2006. ACM.

[16] Microsoft Corp. FxCop home page. http://msdn2.microsoft.com/en-us/library/bb429476(vs.80).aspx.

[17] Roger F. Crew. Astlog: A language for examining abstract syntax trees. In *USENIX Conference on Domain Specific Languages*, pages 229–241, Santa Barbara, 1997.

[18] Oege de Moor, Damien Sereni, Pavel Avgustinov, and Mathieu Verbaere. Type inference for datalog and its application to query optimisation. In *PODS '08: Proceedings of the twenty-seventh ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 291–300, New York, NY, USA, 2008. ACM.

[19] Oege de Moor, Mathieu Verbaere, Elnar Hajiyev, Pavel Avgustinov, Torbjörn Ekman, Neil Ongkingco, Damien Sereni, and Julian Tibble. Keynote address:.QL for source code analysis. In *IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 3–16. IEEE, 2007.

[20] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java Language Specification*. Addison-Wesley, 2000. Second edition.

[21] Georg Gottlob, Erich Grädel, and Helmut Veith. Datalog LITE: a deductive query language with linear time model checking. *ACM Trans. Comput. Logic*, 3(1):42–79, 2002.

[22] Chris Grindstaff. FindBugs, Part 2: Writing custom detectors. http://www.ibm.com/developerworks/java/library/j-findbug2/.

[23] Elnar Hajiyev, Mathieu Verbaere, and Oege de Moor. Codequest: Scalable source code queries with datalog. In Dave Thomas, editor, *ECOOP'06: Proceedings of the 20th European Conference on Object-Oriented Programming*, volume 4067 of *Lecture Notes in Computer Science*, pages 2–27, Berlin, Germany, 2006. Springer.

[24] Bruno Harbulot. Separating concerns in scientific software using aspect-oriented programming. PhD Dissertation, School of Computer Science, University of Manchester, UK, 2006.

[25] Bruno Harbulot and John R. Gurd. A join point for loops in aspectj. In *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development*, pages 63–74, New York, NY, USA, 2006. ACM.

[26] Erik Hilsdale and Jim Hugunin. Advice weaving in AspectJ. In *Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 26–35. ACM Press, 2004.

[27] David Hovemeyer and William Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12):92–106, 2004.

[28] Andrew Hunt and David Thomas. *The Pragmatic Programmer*. Addison-Wesley, 2000.

[29] Neil Immerman. Relational queries computable in polynomial time (extended abstract). In *ACM Symposium on Theory of Computing*, pages 147–152, 1982.

[30] Mizuho Iwaihara and Yusaku Inoue. Bottom-up evaluation of logic programs using binary decision diagrams. In *ICDE '95:*

Proceedings of the Eleventh International Conference on Data Engineering, pages 467–474, Washington, DC, USA, 1995. IEEE Computer Society.

[31] Doug Janzen and Kris De Volder. Navigating and querying code without getting lost. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 178–187, New York, NY, USA, 2003. ACM.

[32] JBoss Inc. JBoss AOP. http://www.jboss.org.

[33] Johnson, S. C. LINT : A C program checker. In *UNIX Programmer Manual*. BELL Labs., 7^{th} edition, 1979.

[34] Karl Lieberherr and David H. Lorenz and Pengcheng Wu. A Case for Statically Executable Advice: checking the Law of Demeter with AspectJ. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 40–49. ACM Press, 2003.

[35] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mike Kersten, Jeffrey Palm, and William Griswold. An Overview of AspectJ. In Jorgen Knudsen, editor, *European Conference on Object-Oriented Programming*, pages 327–353, Budapest, 2001. Springer Verlag.

[36] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming*, pages 220–242. Springer Verlag, 1997.

[37] David Lacey. Program transformation using temporal logic specifications. PhD thesis, Oxford University Computing Laboratory, 2003.

[38] David Lacey, Neil D. Jones, Eric Van Wyk, and Carl Christian Frederiksen. Compiler optimization correctness by temporal logic. *Higher Order Symbol. Comput.*, 17(3):173–206, 2004.

[39] Monica S. Lam, John Whaley, V. Benjamin Livshits, Michael C. Martin, Dzintars Avots, Michael Carbin, and Christopher Unkel. Context-sensitive program analysis as database queries. In *PODS '05: Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 1–12, New York, NY, USA, 2005. ACM.

[40] Ondřej Lhoták and Laurie Hendren. Jedd: a bdd-based relational extension of java. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 158–169, New York, NY, USA, 2004. ACM.

[41] Karl J. Lieberherr and Ian Holland. Assuring good style for object-oriented programs. *IEEE Software*, pages 38–48, September 1989.

[42] Karl J. Lieberherr and Ian Holland. Formulations and Benefits of the Law of Demeter. *SIGPLAN Notices*, 24(3):67–78, March 1989.

[43] Boon Thau Loo, Joseph M. Hellerstein, Ion Stoica, and Raghu Ramakrishnan. Declarative routing: extensible routing with declarative queries. In *SIGCOMM '05: Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 289–300, New York, NY, USA, 2005. ACM.

[44] Boon Thau Loo, Joseph M. Hellerstein, Ion Stoica, and Raghu Ramakrishnan. Declarative routing: extensible routing with

declarative queries. In *SIGCOMM '05: Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 289–300, New York, NY, USA, 2005. ACM.

[45] Martin Aeschlimann and others. Eclipse AST View home page. http://www.eclipse.org/jdt/ut/astview.

[46] Hidehiko Masuhara, Gregor Kiczales, and Chris Dutchyn. Compilation and optimization model for aspect-oriented programs. In *Proceedings of Compiler Construction (CC2003)*, pages 46–60. LNCS, 2003.

[47] David McAllester. On the complexity analysis of static analyses. *J. ACM*, 49(4):512–537, 2002.

[48] Edward McCormick and Kris De Volder. Jquery: finding your way through tangled code. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 9–10, New York, NY, USA, 2004. ACM.

[49] Clint Morgan, Kris De Volder, and Eric Wohlstadter. A static aspect language for checking design rules. In *AOSD '07: Proceedings of the 6th international conference on Aspect-oriented software development*, pages 63–72, New York, NY, USA, 2007. ACM.

[50] Luis Daniel Benavides Navarro, Mario Südholt, Wim Vanderperren, Bruno De Fraine, and Davy Suvée. Explicitly distributed aop using awed. In *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development*, pages 51–62, New York, NY, USA, 2006. ACM.

[51] Muga Nishizawa, Shigeru Chiba, and Michiaki Tatsubori. Remote pointcut: a language construct for distributed aop. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 7–15, New York, NY, USA, 2004. ACM.

[52] Xinming Ou. A logic-programming approach to network security analysis. PhD Dissertation, Princeton University, September 2005.

[53] William Pugh et al. FindBugs home page at sourceforge. http://findbugs.sourceforge.net.

[54] Hridesh Rajan and Kevin Sullivan. Eos: instance-level aspects for integrated system design. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 297–306, New York, NY, USA, 2003. ACM Press.

[55] T. Reps. Demand interprocedural program analysis using logic databases. 1994.

[56] Kouhei Sakurai, Hidehiko Masuhara, Naoyasu Ubayashi, Saeko Matsuura, and Seiichi Komiya. Association aspects. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 16–25, New York, NY, USA, 2004. ACM.

[57] Sun Microsystems, Inc. The official document for class java.lang.Object. http://java.sun.com/j2se/1.4.2/docs/api/java/lang/Object.html.

[58] AspectC++ Team. AspectC++ home page. http://www.aspectc.org. Continuously updated.

[59] AspectJ Team. AspectJ home page. http://www.eclipse.org/aspectj. Continuously updated.

[60] The Eclipse JDT team. Eclipse Java Development Tools home page. http://www.eclipse.org/jdt/.

[61] Anton Dubrau Toheed Aslam, Jesse Doherty and Laurie Hendren. Aosd2010 paper - aspectmatlab: An aspect-oriented scientific programming language. In *AOSD '10:Proceedings of 9th International Conference on Aspect-Oriented Software Development*, March 2010. To appear.

[62] J. D. Ullman. Bottom-up beats top-down for datalog. In *PODS '89: Proceedings of the eighth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 140–149, New York, NY, USA, 1989. ACM.

[63] J. D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume II edition. Computer Science Press, 1989.

[64] Moshe Y. Vardi. The complexity of relational query languages (extended abstract). In *STOC '82: Proceedings of the fourteenth annual ACM symposium on Theory of computing*, pages 137–146, New York, NY, USA, 1982. ACM.

[65] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 131–144, New York, NY, USA, 2004. ACM.

[66] Pengcheng Wu and Karl Lieberherr. Shadow programming: Reasoning about programs using lexical join point information. In *Proceedings of the 4th International Conference on Generative Programming and Component Engineering (GPCE'05)*, 2005.

[67] Xiangyu Zhang, Rajiv Gupta, and Youtao Zhang. Efficient forward computation of dynamic slices using reduced ordered binary decision diagrams. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 502–511, Washington, DC, USA, 2004. IEEE Computer Society.

CHAPTER 8

Appendix

8.1 FxCop rule implementations

Listing 8.1: Abstract types should not have constructors

```
Declarations
pointcut AbstractClassHasConstructors(jp:SH) .
Definitions
AbstractClassHasConstructors(ce) :- ConstructorExec(ce,sig),
                            Signature(sig,t,_,_), TypeInfo(t,_),
                            IsClass(t), TypeModifiers(t,m),
                            ModIsAbstract(m).
```

Listing 8.2: There should be no empty interface

```
Declarations
emptyinterfacetype(t:T) output .
Definitions
emptyinterfacetype(t) :- TypeInfo(t,_), IsInterface(t),
                 !DeclaresMethod(t,_,_,_,_) .
```

Listing 8.3: Avoid excessive type parameters on generic types

```
Declarations
ExcessiveTypeParam(t:T) output .
Definitions
ExcessiveTypeParam(t) :- TypeInfo(t,_), IsGeneric(t),
                         TypeParameter(t,pos,_), pos > 5 .
```

Listing 8.4: Do not catch general exceptions

```
Declarations
pointcut GeneralException(jp:SH) .
Definitions
GeneralException(jp) :- MethodExec(m,_), ExceptionHandler(jp,t,m),
                        TypeInfo(t,"java.lang.Exception") .
GeneralException(jp) :- MethodExec(m,_), ExceptionHandler(jp,t,m),
                        TypeInfo(t,"java.lang.Throwable") .
```

Listing 8.5: Avoid having static members in a generic type

```
Declarations
StaticInGeneric(t:T) output .
Definitions
StaticInGeneric(t) :- TypeInfo(t,_), IsGeneric(t),
                      DeclaresField(t,_,_,m), ModIsStatic(m).
StaticInGeneric(t) :- TypeInfo(t,_), IsGeneric(t),
                      DeclaresMethod(t,_,_,m,_), ModIsStatic(m).
```

Listing 8.6: Avoid having protected members in a final class

```
Declarations
TypeHasProtectedInFinal(t:T) output .
Definitions
TypeHasProtectedInFinal(t) :- TypeInfo(t,_),TypeModifiers(t,m),
                              ModIsFinal(m), DeclaresField(t,_,_,m2),
                              ModIsProtected(m2).
```

Listing 8.7: Exceptions should be public

```
Declarations
ExceptionsNotPublic(t:T) output .
SuperOrSelfTypePublic(t:T) .
Definitions
ExceptionsNotPublic(t) :- TypeInfo(t,_), IsClass(t), SuperType(t,et),
                          TypeInfo(et,"java.lang.Exception"),
                          !SuperOrSelfTypePublic(t) .
SuperOrSelfTypePublic(t) :- TypeInfo(t,_), TypeModifiers(t,m),
                            ModIsPublic(m) .
SuperOrSelfTypePublic(t) :- SuperType(t,tsup),
                            SuperOrSelfTypePublic(tsup) .
```

Listing 8.8: Avoid having visible instance fields

```
Declarations
TypeWithVisibleInstanceField(t:T) output .
Definitions
TypeWithVisibleInstanceField(t) :- TypeInfo(t,_), IsClass(t),
                                   DeclaresField(t,_,_,m),
                                   ModIsPublic(m), !ModIsStatic(m) .
```

8.2 FindBugs rule implementations

Listing 8.9: Class implements Cloneable but does not define clone method

```
Declarations
ClonableClassNoClone(t:T) output .
HasCloneMethod(t:T) .
Definitions
ClonableClassNoClone(t) :- TypeInfo(t,_), Implements(t,ci),
                           TypeInfo(ci,"java.lang.Cloneable"),
                           !HasCloneMethod(t) .
HasCloneMethod(t) :- MethodExec(_,sig), Signature(sig,t,"clone",_) .
```

Listing 8.10: Clone method does not call super.clone()

```
Declarations
pointcut CloneMethodNoSuperClone(jp:SH) .
HasCloneCallOnSuperType(e:SH, t:T) .
Definitions
CloneMethodNoSuperClone(e) :- MethodExec(e,sig),
                              Signature(sig,t,"clone",_),
                              !HasCloneCallOnSuperType(e,t) .
HasCloneCallOnSuperType(e,t) :- MethodCall(_,sig,e),
                          Signature(sig,t2,"clone",_),SuperType(t,t2) .
```

Listing 8.11: Class defines clone method without implementing Cloneable

```
Declarations
pointcut CloneMethodNonCloneable(jp:SH) .
ImplementCloneable (t:T) .
Definitions
CloneMethodNonCloneable(e) :- MethodExec(e,sig),
                              Signature(sig,t,"clone",_),
                              !ImplementCloneable(t) .
ImplementCloneable(t) :- Implements(t,ci),
                         TypeInfo(ci,"java.lang.Cloneable") .
```

Listing 8.12: Class defines covariant compareTo

```
Declarations
pointcut covariantcompareto(jp:SH) .
Definitions
covariantcompareto(e) :- MethodExec(e,sig),
                         Signature(sig,_,"compareTo",_),
                         CodeSignature(sig,psig),
                         CodeSignatureNumParams(psig,1),
                         CodeSignatureParam(psig,0,tp),
                         !TypeInfo(tp,"java.lang.Object") .
```

Listing 8.13: Method might drop exception

```
Declarations
pointcut dropExceptions(jp:SH) .
Throwing(sig:Sig,et:T).
CalleeMayThrow(e:SH,et:T).
Catches(e:SH,et:T).
Definitions
dropExceptions(e) :- MethodExec(e,sig), CalleeMayThrow(e,et),
                     Signature(sig,_,_,_), !Throwing(sig,et),
                     !Catches(e,et) .
Catches(e,et) :- ExceptionHandler(_,et,e) .
Throwing(sig,et) :- Throws(sig,_,et) .
CalleeMayThrow(e,et) :- MethodCall(_,sig,e),Signature(sig,_,_,_),
                        Throwing(sig,et).
```

Listing 8.14: Method might ignore exception

```
Declarations
pointcut ignoreExceptions(jp:SH) .
Throwing(sig:Sig,et:T).
CalleeMayThrow(e:SH,et:T).
Catches(e:SH,et:T).
Definitions
ignoreExceptions(e) :- MethodExec(e,sig), CalleeMayThrow(e,et),
                    Signature(sig,_,_,_),!Catches(e,et) .
Catches(e,et) :- ExceptionHandler(_,et,e) .
Throwing(sig,et) :- Throws(sig,_,et) .
CalleeMayThrow(e,et) :- MethodCall(_,sig,e),Signature(sig,_,_,_),
                      Throwing(sig,et).
```

Listing 8.15: Do not use removeAll to clear a collection

```
Declarations
pointcut removesAll(jp:SH) .
Definitions
removesAll(c) :- MethodCall(c,sig,_), Signature(sig,t,"removesAll",_),
               SuperType(t,ct),TypeInfo(ct,"java.util.Collection") .
```

Listing 8.16: Do not call a few dangerous methods on System class

```
Declarations
pointcut dangerousCalls(jp:SH) .
dangerousMethodNames(n:S) .
Definitions
dangerousCalls(c) :- MethodCall(c,sig,_), Signature(sig,t,name,_),
                   TypeInfo(t,"java.lang.System"),
                   dangerousMethodNames(name).
dangerousMethodNames("exit").
dangerousMethodNames("runFinalizersOnExit") .
```

Listing 8.17: Class defines compareTo but uses Object.equals()

```
Declarations
pointcut comparesToNoEquals(jp:SH) .
hasEquals (t:T) .
Definitions
comparesToNoEquals(e) :- MethodExec(e,sig),
                         Signature(sig,t,"compareTo",_),
                         !hasEquals(t) .
hasEquals(t) :- MethodExec(_,sig), Signature(sig,t,"equals",_) .
```

Covariant equals method defined

This is very similar to Listing 8.12.

Listing 8.18: Explicit invocation of finalizer should be prohibited

```
Declarations
pointcut callsFinalizer(jp:SH) .
Definitions
callsFinalizer(c) :- MethodCall(c,sig,_),
                     Signature(sig,_,"finalizer",_).
```

8.3 Predicate definitions for Abstract Syntax Tree based model

Listing 8.19: AST Predicates

```
######## Domains
ID #Domain of AST node/binding IDs
Z #Domain of integers
S #Domain of strings
B #Domain of booleans
K #Domain of type kinds
MOD #Domain of modifiers

#Declarations
 CompilationUnit(ind:ID)
 CompilationUnit_IMPORTS(parent:ID, pos:Z, child:ID, length:Z)
 CompilationUnit_TYPES(parent:ID, pos:Z, child:ID, length:Z)
 TypeDeclaration(ind:ID, TypeBinding:ID, isInterface:B)
 TypeDeclaration_MODIFIERS(parent:ID, pos:Z, child:ID, length:Z)
 TypeDeclaration_NAME(parent:ID, child:ID)
 TypeDeclaration_TYPE_PARAMETERS(parent:ID, pos:Z, child:ID, length:Z)
 TypeDeclaration_SUPER_INTERFACE_TYPES(parent:ID, pos:Z, child:ID,
                                       length:Z)
 TypeDeclaration_BODY_DECLARATIONS(parent:ID, pos:Z, child:ID, length:Z)
 TypeBinding(ind:ID, name:S, key:S, qualifiedName:S, kind:K, elementType:ID,
            componentType:ID, dimensions:Z, PACKAGE:ID, declaringClass:ID,
            declaringMeth:ID, modifiers:MOD, SUPERCLASS:ID,
            implementedInterfaces:ID, declaredTypes:ID)
 ElementType(ind:ID)
 PackageBinding(ind:ID, name:S, key:S, unamed:B)
 SimpleName(ind:ID, TypeBinding:ID, kindBinding:ID, identifier:S)
 MethodDeclaration(ind:ID, MethodBinding:ID, constr:B, extra_dimensions:S)
 MethodDeclaration_MODIFIERS(parent:ID, pos:Z, child:ID, length:Z)
 MethodDeclaration_TYPE_PARAMETERS(parent:ID, pos:Z, child:ID, length:Z)
 MethodDeclaration_RETURN_TYPE2(parent:ID, child:ID)
 MethodDeclaration_NAME(parent:ID, child:ID)
```

```
MethodDeclaration_PARAMETERS(parent:ID, pos:Z, child:ID, length:Z)
MethodDeclaration_THROWN_EXCEPTIONS(parent:ID, pos:Z, child:ID, length:Z)
MethodBinding(ind:ID, name:S, key:S, isConstr:B, defaultConstr:B,
             declaringClass:ID, returnType:ID, modifiers:MOD,
             parameterTypes:ID, varargs:B, exceptionTypes:ID)
ParameterTypes(ind:ID, pos:Z, TypeBinding:ID)
Modifier(ind:ID, keyword:S)
PrimitiveType(ind:ID, TypeBinding:ID, typecode:S)
SingleVariableDeclaration(ind:ID, VariableBinding:ID, VARARGS:B,
                        extra_dimensions:S)
SingleVariableDeclaration_MODIFIERS(parent:ID, pos:Z, child:ID, length:Z)
SingleVariableDeclaration_TYPE(parent:ID, child:ID)
SingleVariableDeclaration_NAME(parent:ID, child:ID)
VariableBinding(ind:ID, name:S, key:S, isField:B, isEnumConst:B, isParam:B,
               variableId:Z, modifiers:MOD, TYPE:ID, declaringClass:ID,
               declaringMeth:ID)
FieldDeclaration(ind:ID)
FieldDeclaration_MODIFIERS(parent:ID, pos:Z, child:ID, length:Z)
FieldDeclaration_TYPE(parent:ID, child:ID)
FieldDeclaration_FRAGMENTS(parent:ID, pos:Z, child:ID, length:Z)
VariableDeclarationFragment(ind:ID, VariableBinding:ID, extra_dimensions:S)
VariableDeclarationFragment_NAME(parent:ID, child:ID)
VariableDeclarationFragment_INITIALIZER(parent:ID, child:ID)
NumberLiteral(ind:ID, TypeBinding:ID, token:S)
ImplementedInterfaces(ind:ID, pos:Z, TypeBinding:ID)
SimpleType(ind:ID, TypeBinding:ID)
SimpleType_NAME(parent:ID, child:ID)
MethodDeclaration_BODY(parent:ID, child:ID)
Block(ind:ID)
Block_STATEMENTS(parent:ID, pos:Z, child:ID, length:Z)
ExpressionStatement(ind:ID)
ExpressionStatement_EXPRESSION(parent:ID, child:ID)
Assignment(ind:ID, TypeBinding:ID, operator:S)
Assignment_LEFT_HAND_SIDE(parent:ID, child:ID)
Assignment_RIGHT_HAND_SIDE(parent:ID, child:ID)
InfixExpression(ind:ID, TypeBinding:ID, operator:S)
```

```
InfixExpression_LEFT_OPERAND(parent:ID, child:ID)
InfixExpression_RIGHT_OPERAND(parent:ID, child:ID)
InfixExpression_EXTENDED_OPERANDS(parent:ID, pos:Z, child:ID, length:Z)
ReturnStatement(ind:ID)
ReturnStatement_EXPRESSION(parent:ID, child:ID)
DeclaredTypes(ind:ID, pos:Z, TypeBinding:ID)
QualifiedName(ind:ID, TypeBinding:ID, kindBinding:ID)
QualifiedName_QUALIFIER(parent:ID, child:ID)
QualifiedName_NAME(parent:ID, child:ID)
MethodInvocation(ind:ID, TypeBinding:ID, MethodBinding:ID)
MethodInvocation_EXPRESSION(parent:ID, child:ID)
MethodInvocation_TYPE_ARGUMENTS(parent:ID, pos:Z, child:ID, length:Z)
MethodInvocation_NAME(parent:ID, child:ID)
MethodInvocation_ARGUMENTS(parent:ID, pos:Z, child:ID, length:Z)
IfStatement(ind:ID)
IfStatement_EXPRESSION(parent:ID, child:ID)
IfStatement_THEN_STATEMENT(parent:ID, child:ID)
IfStatement_ELSE_STATEMENT(parent:ID, child:ID)
PostfixExpression(ind:ID, TypeBinding:ID, operator:S)
PostfixExpression_OPERAND(parent:ID, child:ID)
ForStatement(ind:ID)
ForStatement_INITIALIZERS(parent:ID, pos:Z, child:ID, length:Z)
ForStatement_EXPRESSION(parent:ID, child:ID)
ForStatement_UPDATERS(parent:ID, pos:Z, child:ID, length:Z)
ForStatement_BODY(parent:ID, child:ID)
VariableDeclarationExpression(ind:ID, TypeBinding:ID)
VariableDeclarationExpression_MODIFIERS(parent:ID, pos:Z, child:ID,
                                        length:Z)
VariableDeclarationExpression_TYPE(parent:ID, child:ID)
VariableDeclarationExpression_FRAGMENTS(parent:ID, pos:Z, child:ID,
                                        length:Z)
StringLiteral(ind:ID, TypeBinding:ID, value:S)
WhileStatement(ind:ID)
WhileStatement_EXPRESSION(parent:ID, child:ID)
WhileStatement_BODY(parent:ID, child:ID)
SwitchStatement(ind:ID)
```

```
SwitchStatement_EXPRESSION(parent:ID, child:ID)
SwitchStatement_STATEMENTS(parent:ID, pos:Z, child:ID, length:Z)
SwitchCase(ind:ID)
SwitchCase_EXPRESSION(parent:ID, child:ID)
BreakStatement(ind:ID)
ThrowStatement(ind:ID)
ThrowStatement_EXPRESSION(parent:ID, child:ID)
ClassInstanceCreation(ind:ID, TypeBinding:ID, MethodBinding:ID)
ClassInstanceCreation_TYPE_ARGUMENTS(parent:ID, pos:Z, child:ID, length:Z)
ClassInstanceCreation_TYPE(parent:ID, child:ID)
ClassInstanceCreation_ARGUMENTS(parent:ID, pos:Z, child:ID, length:Z)
TypeDeclaration_SUPERCLASS_TYPE(parent:ID, child:ID)
ArrayType(ind:ID, TypeBinding:ID)
ArrayType_COMPONENT_TYPE(parent:ID, child:ID)
ArrayCreation(ind:ID, TypeBinding:ID)
ArrayCreation_TYPE(parent:ID, child:ID)
ArrayCreation_DIMENSIONS(parent:ID, pos:Z, child:ID, length:Z)
TryStatement(ind:ID)
TryStatement_BODY(parent:ID, child:ID)
TryStatement_CATCH_CLAUSES(parent:ID, pos:Z, child:ID, length:Z)
NullLiteral(ind:ID, TypeBinding:ID)
CatchClause(ind:ID)
CatchClause_EXCEPTION(parent:ID, child:ID)
CatchClause_BODY(parent:ID, child:ID)
SuperMethodInvocation(ind:ID, TypeBinding:ID, MethodBinding:ID)
SuperMethodInvocation_TYPE_ARGUMENTS(parent:ID, pos:Z, child:ID, length:Z)
SuperMethodInvocation_NAME(parent:ID, child:ID)
SuperMethodInvocation_ARGUMENTS(parent:ID, pos:Z, child:ID, length:Z)
BooleanLiteral(ind:ID, TypeBinding:ID, bv:B)
MethodDeclaration_JAVADOC(parent:ID, child:ID)
Javadoc(ind:ID)
Javadoc_TAGS(parent:ID, pos:Z, child:ID, length:Z)
TagElement(ind:ID, tagName:S)
TagElement_FRAGMENTS(parent:ID, pos:Z, child:ID, length:Z)
VariableDeclarationStatement(ind:ID)
VariableDeclarationStatement_MODIFIERS(parent:ID, pos:Z, child:ID,
```

```
140                                     length:Z)
    VariableDeclarationStatement_TYPE(parent:ID, child:ID)
142 VariableDeclarationStatement_FRAGMENTS(parent:ID, pos:Z, child:ID,
                                        length:Z)
144 ThisExpression(ind:ID, TypeBinding:ID)
    DoStatement(ind:ID)
146 DoStatement_EXPRESSION(parent:ID, child:ID)
    DoStatement_BODY(parent:ID, child:ID)
148 LineComment(ind:ID)
    HasChild(parent:ID, child:ID)
150 NumOfElements(id:ID, num:Z)
    IsArray(k:K)
152 IsCapture(k:K)
    IsNullType(k:K)
154 IsPrimitive(k:K)
    IsTypeVariable(k:K)
156 IsWildcardType(k:K)
    IsAnnotation(k:K)
158 IsClass(k:K)
    IsInterface(k:K)
160 IsEnum(k:K)
    IsAbstract(m:MOD)
162 IsFinal(m:MOD)
    IsNative(m:MOD)
164 IsPrivate(m:MOD)
    IsProtected(m:MOD)
166 IsPublic(m:MOD)
    IsStatic(m:MOD)
168 IsVolatile(m:MOD)
    IsSynchronized(m:MOD)
```