Optimizing JML Feature Compilation in Ajmlc Using Aspect-Oriented Refactorings

Henrique Rebêlo[1], Ricardo Lima[1], Márcio Cornélio[2], Gary T. Leavens[3], Alexandre Mota[1], César Oliveira[1]

[1]Informatics Center — Federal University of Pernambuco
Caixa Postal 7851, 50740-540 — Recife — PE — Brazil

[2]Department of Computing and Systems — University of Pernambuco
Rua Benfica, 455, Madalena, 50720-001 — Recife — PE — Brazil

[3]School of Electrical Engineering and Computer Science — University of Central Florida
4000 Central Florida Blvd. — Orlando — FL — USA

`{hemr,rmfl,acm,calo}@cin.ufpe.br, marcio@dsc.upe.br, leavens@eecs.ucf.edu`

Abstract. *In previous work we presented a new JML compiler, ajmlc, which generates aspects that enforce preconditions, postconditions, and invariants. Although this compiler provides benefits of source-code modularity and good bytecode size and running time, there is still a need for optimization of the generated code's bytecode size and running time. To do this optimization while preserving the semantics of the resulting code, we use refactorings based on AspectJ programming laws. To this end we present optimization refactorings and an empirical analysis showing the resulting improvements.*

1. Introduction

Restructuring an aspect-oriented [Kiczales 1996] program is a useful activity known as refactoring. Refactoring in the object-oriented sense [Opdyke 1992, Roberts 1999, Fowler et al. 1999] involves changes such as moving attributes and methods between classes or splitting one class into several classes. Refactorings preserve a program's observable behavior while allowing one to improve its modularity, decrease its size, etc. Our goal is to apply aspect-oriented refactorings, automatically, in an optimizing compiler that generates aspect-oriented code. In such a compiler, the consequences of incorrect transformations can be greatly amplified. Thus the problem is to ensure that such refactorings are very trustworthy, that is, that they correctly preserve the program's observable behavior.

Our approach to solving this problem is to design refactorings using aspect-oriented programming laws inspired by those proposed by Cole and Borba [Cole and Borba 2005]. Our compiler, ajmlc, generates code written in AspectJ [Kiczales et al. 2001], a general purpose aspect-oriented extension to Java. To optimize this generated AspectJ code, we need programming laws that apply to AspectJ. For this we draw on the work of Cole and Borba [Cole and Borba 2005]. Their laws establish how to restructure AspectJ code, by adding or removing AspectJ constructs. We use their laws and have developed others to derive optimizing transformations, which are refactoring rules applied in a particular direction. Soundness of a few of the refactoring transformations follows from the soundness of Cole and Borba's laws, which have already been proven correct [Cole et al. 2005]. However, most of our laws are adaptations of their laws, aimed at optimizing aspect oriented code.

The ajmlc compiler itself was described in a previous work [Rebêlo et al. 2008b]. It takes input written in the Java Modeling Language (JML) [Burdy et al. 2005, Leavens 2006] and generates aspects to check the JML specifications at runtime. Unlike the classical JML compiler, jmlc [Cheon 2003], ajmlc does not use Java's reflection facilities, and thus can also be applied to constrained environments such as Java ME applications. While there are several related works that implement such dynamic contract checking using aspects [Briand et al. 2005, Feldman et al. 2006, Wampler 2006], none of them optimizes the generated aspects. This optimization of generated aspect code is what we demonstrate using ajmlc.

The contributions of this paper are threefold. First, it describes a collection of aspect-oriented laws and refactorings used to restructure AspectJ constructs. Second, the paper details results about the use and the importance of such laws and refactorings in optimizing ajmlc aspects. To better explain the impacts of the optimizations, we provide a case study with four programs. Third, to the best of our knowledge, this is the first work that shows how to optimize assertion checking code. While we present these laws and refactorings using JML, they are independent of JML, and can be used in other AspectJ programs.

This paper is organized as follows. We give an overview of JML in Section 2. After that, in Section 3, we present the proposed aspect-oriented laws and refactorings. In Section 4, we quantify the use and the benefits of the proposed laws and refactorings in a case study involving four programs. In Section 5, we discuss related work and in Section 6, we present our conclusions.

2. An Overview of JML

Java has assertions, but no other built-in support for Design by Contract (DbC). The Java Modeling Language (JML) [Leavens et al. 2006, Leavens 2006] provides DbC support for Java.

JML includes a number of constructs to declaratively specify runtime behavior. Classes are declared by specifying their fields, invariants over those fields, and by specifying the behavior of constructors and methods. (In the following, we refer to both constructors and methods as "methods" when there is no need to distinguish them.) Basic method specifications are written using pre- and postconditions. Such JML specifications are written in Java code files using special comments, as shown in Figure 1. This figure shows a simple JML specification for a class `JMLExample` with a method `div`. The method's contract is composed of a precondition, requiring `b > 0` and a postcondition, ensuring that the method's result is `a / b`.

There are a number of tools that work with JML [Burdy et al. 2005], including the classical JML compiler (jmlc) [Cheon 2003]. Like jmlc, our ajmlc compiler [Rebêlo et al. 2008b] translates JML-annotated Java source code into Java bytecode with automatic runtime checks. Unlike jmlc, ajmlc generates AspectJ code. For example, Figure 2 shows the AspectJ code generated by ajmlc to check the precondition defined in Figure 1 (some details are omitted for simplicity).

```
public class JMLExample {
  //@ requires b > 0;
  //@ ensures \result == a / b;
  public int div(int a, int b) {
    return a/b;
  }
}
```

Figure 1. Example of JML specification.

```
before (C obj, int a, int b) :
 execution(int C.div(int,int))
 within(C) &&
 this(obj) && args(b) {
   boolean rac$b = true;
   rac$b = obj.checkPre$div;
   if(!rac$b){
     throw new
       JMLPreconditionError("");
   }
 }

public boolean C.checkPre$div(int a, int b) {
  return b > 0;
}
```

Figure 2. The aspect code to check `div`'s precondition defined in Figure 1.

3. Laws and Refactorings

For establishing a systematic and rigorous basis for optimization via program transformation, we use algebraic laws of programming [Hoare et al. 1987] to design code optimizers [Sampaio 1997]. We illustrate the use of the algebraic approach by considering two programming laws [Hoare et al. 1987]: (1) one related to the assignment command, and (2) one related to sequential composition. The former law states that the assignment of a variable's value to itself has no effect. The latter law states that a command **skip**, preceding or following a $stmt$, does not change the effect of the $stmt$.

Law ⟨*void assignment*⟩

$$(x := x) = \mathbf{skip} \qquad \Box$$

Law ⟨*unit-skip*⟩

$$(\mathbf{skip}; stmt) = (stmt; \mathbf{skip}) = stmt \qquad \Box$$

The sequential use of the above laws improves code quality (by making it smaller) and consequently decreases the program's expected execution time, which is our objective. Our refactorings exploit such composition laws, and also exploit AspectJ programming laws [Cole and Borba 2005].

3.1. Aspect-Oriented Refactoring

Since ajmlc generates aspect-oriented (AO) code, we need to refactor this generated AO code, attempting to increase its performance and to decrease its size. Hannemann [Hannemann et al. 2005] classifies aspect-oriented refactorings into three distinct groups:

1. aspect-aware OO refactorings;
2. refactorings of AO constructs;
3. refactorings of crosscutting concerns.

This paper only focuses on the second group, refactorings of AO constructs.

3.2. Deriving AspectJ refactorings using programming laws

Several works have been identified common transformations for aspect-oriented programs [Monteiro and Fernandes 2005, Hannemann et al. 2005, Laddad 2006, Iwamoto and Zhao 2003], mostly in AspectJ. Nevertheless, such works lack support for assuring that the transformations preserve behavior and are indeed refactorings. In contrast, Cole and Borba describe a set of AspectJ programming laws that give us a basis for proving that the transformations preserve behavior and, therefore, are indeed refactorings [Cole and Borba 2005]. Thus, some of the refactorings we present are derived from, and proven correct using [Cole et al. 2005] their AspectJ programming laws. We also use other refactorings that do not directly correspond to their programming laws. Formally proving the soundness of these new laws is future work.

Notation

The subset of laws and refactorings we present in this paper are written using two boxes written side by side, followed by a **provided** clause. This clause gives conditions, also known as provisos, all of which must be true for the the law or refactoring to be correctly applied. The notation "(→)" introduces each proviso, and indicates a proviso that must be satisfied when applying the rule from left-to-right. (We present all our laws and refactoring rules as one-way left-to-right rules, since we only use them for optimization [Sampaio 1997].)

Laws and refactoring rules

The first law we present (**Law** 1) allows us to remove an empty privileged aspect, provided that A is not referenced in ts; the set of type declarations (classes and aspects). We use $\mathbf{paspect}$ to denote a privileged aspect declaration for simplicity. We easily derive this law by applying Cole and Borba's laws ⟨*make aspect privileged*⟩ and ⟨*add empty aspect*⟩ [Cole and Borba 2005]. Both laws are applied from right-to-left. Note that the derived law (Law 1) is applied from left-to-right, as assumed in the notation.

Law 1. $\langle$*remove empty privileged aspect*$\rangle$

ts
paspect A {
}
	$=$	ts

provided
($\rightarrow$) A is not referenced from ts. $\square$

Law 1 is useful in ajmlc optimization when no JML annotations are provided (or when an empty class is being compiled), since for such code ajmlc generates an empty privileged aspect. Note that the classical JML compiler (jmlc) [Cheon 2003] always generates 11.0 KB of source code instrumentation, which it compiles to 5.93 KB of bytecode instrumentation, even for empty classes [Rebêlo et al. 2008a].

Law 5 shows a transformation which removes **before** advice when we apply it from left-to-right. We use $\sigma(C.m)$ to denote the signature of method m of class C; its return type and formal parameters are denoted by T and ps, respectively. Moreover, we use $bind(context)$ to denote the list of advice parameters, including the current executing object (represented by $cthis$), which binds the AspectJ advice parameters (**this**, `args`). Additionally, we use the AspectJ designator $\mathbf{within}(C)$ to prevent the **before** advice from applying to executions of method m in subtypes of C. We write $body'[cthis.m']$ to indicate that $body'$ may contain a reference to the method m', having $cthis$ as target.

Law 5. $\langle$*remove before-execution*$\rangle$

```
ts
class C {
  fs
  ms
  T m(ps) {
    body
  }
}
paspect A {
  as
  before(context) :
    exec(σ(C.m)) &&
    within(C) &&
    bind(context) {
    body'[cthis.m']
  }
  T' C.m'(ps) {
    this.exp
  }
}
```

$=$

```
ts
class C {
  fs
  ms
  T m(ps) {
    body
  }
}
paspect A {
  as
  T' C.m'(ps) {
    this.exp
  }
}
```

provided

($\rightarrow$) $\mathbf{before}$ advice does not contribute to execution flow of the affected join point $\sigma(C.m)$, or type C is declared $\mathbf{abstract}$ or it is declared as an $\mathbf{interface}$. $\square$

The proviso states that the $\mathbf{before}$ advice does not add any behavior to the affected method m. Thus, we can remove it. Moreover, we can also remove the $\mathbf{before}$ advice if the declared type is $\mathbf{abstract}$ or if it is an $\mathbf{interface}$. These latter two conditions are valid because the required $\mathbf{within}(C)$ point cut designator does not allow the advice to apply to subtypes, and we cannot instantiate a concrete class when we have an $\mathbf{abstract}$ or an $\mathbf{interface}$ type. Therefore, we always can remove such an advice.

This is the simplest law to remove advice, thus it can be applied to other advice as well. In this way, we can remove other advice by applying in as, which refers to other advice in the left side of the law template. The derivation of this law is also simple. We apply the Law $\langle$*add before execution*$\rangle$ [Cole and Borba 2005, Law 3] from right-to-left. However, this law is slightly different from ours, because it is concerned with OO code transformations into AO code. In this way, our proviso must consider different situations, even though the result is the same advice elimination.

In the context of JML and ajmlc, **Law** 5 is useful when we specify abstract classes or interfaces. So, if we specify a concrete class and, for example, a method has a default precondition (i.e., its precondition is **`true`**), then we can remove the related advice, since the before advice that is generated will not contribute to the execution flow of the affected join point.

The next rule, **Refactoring** 1, is a refactoring that, when applied from left-to-right, inlines the method intertype implementation within $\mathbf{before}$ advice. This transformation is useful because the method intertype is only referenced by one advice. The transformation removes the method intertype and moves its implementation to the advice. The derivation of this refactoring involves two other simple laws. (These new laws are not presented here, since they are simple.) Consider them step by step: (1) apply **Law** 3 $\langle$*replace method intertype reference with method intertype implementation within advice*$\rangle$, replacing all references of the method intertype m' within $\mathbf{before}$ advice with its implementation, and (2) apply **Law** 4 $\langle$*remove method intertype related to advice*$\rangle$, removing the method intertype m'.

Refactoring 1 is useful for ajmlc optimization when, for example, a $\mathbf{before}$ advice is checking a precondition, and this advice references a method intertype with a precondition predicate that is not referenced by any other advice, aspect or class. Since the method intertype is only referenced in one place by the advice, it is useful if ajmlc can eliminate it by using the **Refactoring** 1. This scenario is illustrated in Figure 2, where we can see pieces of code generated by ajmlc. The result of applying refactoring 1 is shown in Figure 3.

Refactoring 1. $\langle$*inline method intertype within before-execution*$\rangle$

```
ts
class C {
  fs
  ms
  T m(ps) {
    body
  }
}
paspect A {
  as
  before(context) :
    exec(σ(C.m)) &&
    within(C) &&
    bind(context) {
    body′[cthis.m′]
  }
  T′ C.m′(ps) {
    this.exp
  }
}
```

=

```
ts
class C {
  fs
  ms
  T m(ps) {
    body
  }
}
paspect A {
  as
  before(context) :
    exec(σ(C.m)) &&
    within(C) &&
    bind(context) {
    body′[cthis.exp]
  }
}
```

provided

$(\rightarrow)$ m is not referenced from C, ts, or as. $\square$

```
before (C obj, int a, int b) :
 execution(int C.div(int,int))
 within(C) &&
 this(obj) && args(b) {
   boolean rac$b = true;
   rac$b = obj.checkPre$div;
   if(!rac$b){
     throw new
       JMLPreconditionError("");
   }
 }

public boolean C.checkPre$div(int a, int b) {
  return b > 0;
}
```

=

```
before (C obj, int a, int b) :
 execution(int C.div(int,int))
 within(C) &&
 this(obj) && args(b) {
   boolean rac$b = true;
   rac$b = b > 0;
   if(!rac$b){
     throw new
       JMLPreconditionError("");
   }
 }
```

Figure 3. Result of the application of Refactoring 1 in the AspectJ code presented in Figure 2.

The template of **Refactoring** 1 shows the transformation concerning a **before** advice, but such a refactoring can deal with other kinds of AspectJ advice [Laddad 2003].

Table 1. Summary of Aspect-Oriented Laws and Refactorings

Laws	Refactorings
1. *remove empty privileged aspect* 2. *move advice body to other advice* 3. *replace method intertype reference with method intertype implementation within advice* 4. *remove method intertype related to advice* 5. *remove before-execution* 6. *remove after-execution returning* 7. *remove after-execution throwing* 8. *remove around-execution* 9. *remove this designator* 10. *remove within designator*	1. *inline method intertype within advice* 2. *merge distinct advice* 3. *split around-execution into after-execution returning and after-execution throwing* 4. *extract aspect method*

Other laws and refactorings

Besides the laws and refactorings presented above, we derived others [1] that greatly helped ajmlc's optimization reduce the result code's execution time. Table 1 lists all of these laws and refactorings [2]. For example, the **Law** 2 ⟨*move advice body to other advice*⟩, enables one to move the implementation of a **`before`** advice to the body of an **`around`** advice (before the call to **`proceed`**). The only proviso is that both pieces of advice must apply to just one join point of just one type.

By using **Law** 2 and **Law** 5, we can derive **Refactoring** 2 ⟨*merge distinct advice*⟩, which enables one, for example, to merge a **`before`** advice into an **`around`** advice. This simplifies the aspect-oriented code by reducing the number of pieces of advice affecting the same join point. After **Law** 2, we apply **Law** 5 to remove the empty **`before`** advice. Since this refactoring is derived using **Law** 2, the advice must apply to just one join point. However, this condition is satisfied often in the context of JML and ajmlc optimization. One common case occurs when there is a static method to which one **`before`** advice applies to check preconditions and one **`around`** advice applies to check postconditions. Since the method is static, the generated advice only applies to it, and there are no subtypes to consider. Hence ajmlc can merge such advice by means of **Refactoring** 2.

Soundness

As mentioned above, programming laws [Hoare et al. 1987] define equivalence between two programs, given that some conditions are respected. However, the proof of the behavior preserving property of programming laws is not trivial. So, the soundness of our laws relies on the proofs of Cole and Borba or follows their proof strat-

[1] In order to see other law and refactoring templates as well as their application in AspectJ code, please refer to [Rebêlo et al. 2009].

[2] Except by **Law** 1, which we derived completely from [Cole and Borba 2005, Cole et al. 2005], the other laws and refactorings are proposed by this work.

Table 2. Laws and Refactorings in the JAccounting, JSpider, Prevayler, and Bomber systems

	JAccounting Qty	JSpider Qty	Prevayler Qty	Bomber Qty
Law 1	28	148	69	5
Law 2	8	9	8	3
Law 3	2	95	40	2
Law 4	5	115	65	2
Law 5	33	27	46	10
Law 6	30	36	57	0
Law 8	11	249	126	20
Law 9	34	7	1	2
Law 10	34	7	1	2
Refactoring 1	2	94	39	2
Refactoring 2	8	9	2	3
Refactoring 3	11	249	126	20

egy [Cole and Borba 2005, Cole et al. 2005]. As our laws are justified by means of compositions of their laws, we rely on their correctness proofs for their laws.

However, there are laws in our work that are not derived from Cole and Borba's. Proving the soundness of these laws using a formal semantics is desirable, thus, as future work we intend to use the same formal semantics to prove that these laws are behavior-preserving transformations. Even though we have some laws that are not yet proved sound, we have informally considered their correctness. This is possible because, compared to refactorings, such laws are much simpler, involving only local changes, and each one concerns only a specific AspectJ construct.

4. Case Study

This section presents the results of a case study involving the JAccounting [3], JSpider [4], Prevayler [5], and Bomber [6] programs. Our proposed laws and refactorings are automatically applied to these four programs by the ajmlc optimizer, and we analyze the benefits of our approach for both programs. Table 2 summarizes how many laws and refactorings are applied by ajmlc's optimizer in each program.

We compiled these four programs, after annotating them with JML, using both the classic JML compiler jmlc [Cheon 2003] and our own ajmlc [Rebêlo et al. 2008b]. For ajmlc we employed two versions: both with and without the laws and the refactorings (optimizations) proposed in this work. Moreover, we used ajmlc with two different weaving processes: the standard AspectJ compiler (ajc), and abc [Avgustinov et al. 2005]. The difference is that the abc weaver itself includes various optimizations.

Our case study considers a Java ME application because ajmlc, unlike jmlc, can

[3]https://jaccounting.dev.java.net.
[4]http://j-spider.sourceforge.net/.
[5]http://www.prevayler.org/.
[6]http://j2mebomber.sourceforge.net.

Table 3. Code size measurements

Application	Original (MB)	Optimized (MB)	Decrease (%)	Application	Original (MB)	Optimized (MB)	Decrease (%)
JAccounting				**JSpider**			
– ISC –				*– ISC –*			
jmlc	5.46	-	-	jmlc	10.50	-	-
ajmlc	1.32	1.10	16.66	ajmlc	2.15	1.82	15.34
– Bytecode –				*– Bytecode –*			
jmlc	1.95	-	-	jmlc	4.18	-	-
ajmlc(ajc)	3.70	2.29	38.10	ajmlc(ajc)	6.96	4.54	34.77
ajmlc(abc)	1.32	0.90	31.81	ajmlc(abc)	2.54	1.75	31.10
Prevayler				**Bomber**			
– ISC –				*– ISC –*			
jmlc	4.54	-	-	jmlc	-	-	-
ajmlc	0.99	0.83	16.16	ajmlc	0.89	0.79	11.23
– Bytecode –				*– Bytecode –*			
jmlc	1.45	-	-	jmlc	-	-	-
ajmlc(ajc)	3.33	2.10	36.93	ajmlc(ajc)	2.15	1.40	34.88
ajmlc(abc)	1.20	0.87	27.50	ajmlc(abc)	0.84	0.60	28.57

compile and run Java ME applications [Rebêlo et al. 2008b]. This Java ME application is the Bomber program. It is a simple software product line game based on Java ME MIDP 2.0.

Code size and performance statistics

In this case study we gathered some measurements that demonstrated an improvement in both code size and performance of the generated AO code, as optimized. Tables 3 and 4 present the results that we obtained by assessing the proposed refactorings. In Table 3 instrumented source code size is denoted by *ISC*, and instrumented bytecode size is denoted by *Bytecode*. Code size is measured in megabytes (MB). In Table 4 execution time is measured in milliseconds (msec).

In relation to code size (see Table 3), we observed that the optimized ajmlc aspect code is smaller than the non-optimized code, both in the size of the instrumented source code and the corresponding bytecode. It is worth noting that the effect on the bytecode is relevant, producing far smaller class files, when the abc weaver is employed. It is also clear that ajmlc produces smaller instrumented source code than jmlc, even without any of our optimizations. Nevertheless, after compilation, we observed that the ajmlc compiler has smaller bytecode instrumentation than jmlc only when the ajmlc output is both optimized and compiled using the abc weaver. This indicates that the compilation techniques for aspect oriented programs are still in a stage of evolution.

Concerning running time (see Table 4), we observed that the optimized ajmlc produced code that executes faster than the non-optimized version. As shown in that table, the running time is greatly reduced when the optimized ajmlc employs the abc weaver. Additionally, as noted, the running time of the ajmlc aspects code is faster than the jmlc code, even without using our optimizations. Such bad performance in jmlc is due to many reflective calls in the jmlc generated code. Note that the method JSpider/translate

Table 4. Running time measurements

Method	Original (msec)			Optimized (msec)		Decrease (%)	
	jmlc	ajmlc ajc	ajmlc abc	ajmlc ajc	ajmlc abc	ajmlc ajc	ajmlc abc
JAccounting/getCreated	0.33	0.06	0.05	0.04	0.03	33.33	40.00
JAccounting/getCompanyKey	0.33	0.06	0.05	0.03	0.04	50.00	20.00
JAccounting/perform2	6.9	5.78	5.75	4.97	4.90	14.01	14.78
JSpider/createTool	25.22	35.81	36.02	21.32	26.28	40.46	27.04
JSpider/createURL	6.77	11.55	6.17	4.43	4.33	61.64	29.82
JSpider/translate	7.53	**10.48**	7.31	**27.98**	7.05	**-62.54**	3.55
Prevayler/createAccount	0.18	0.044	0.042	0.043	0.039	2.27	7.14
Prevayler/deleteAccount	0.48	0.11	0.10	0.09	0.08	18.18	20.00
Prevayler/findAccount	0.40	0.11	0.09	0.09	0.08	18.18	11.11
Bomber/handle	-	3.47	0.07	2.97	0.04	14.40	42.85
Bomber/getRadius	-	3.97	0.06	3.15	0.04	20.65	33.33
Bomber/getDamage	-	3.53	0.05	3.32	0.03	5.94	40.00

presents a worst performance with the ajc after applying the optimizations. For this particular method, we split the **`around`** advice into two advices: **`after returning`**, and **`after throwing`**. We are still investigating what is causing such negative impact in the performance. But we suspect that although ajc generates a code with better performance for the **`around`** advice, when compared with the two types of **`after`** advices, the resulting code size is bigger. This is not observed in the abc because it implements optimizations specifically designed for decrease the code size of the **`around`** advice. Note that we did not measure the execution time of methods compiled with jmlc in the Bomber program. This is due to the lack of support for reflection and other Java SE features by Java ME applications [Rebêlo et al. 2008b]. Thus, we cannot execute jmlc generated code for the Bomber program with the Java ME API that it uses.

5. Related Work

We discuss related work in the context of refactorings for object-oriented and aspect-oriented programs.

The seminal work on the formalization of refactoring was presented by Opdyke [Opdyke 1992]. His work focuses on object-oriented refactoring, whereas our work focuses on aspect-oriented refactorings. As with our work, the main importance of Opdyke's work is not only the identification of refactorings, but also the definition of the preconditions that are required to apply each refactoring without changing the program's behavior.

Cole and Borba [Cole and Borba 2005] present aspect-oriented programming laws that can be used to derive refactorings for AspectJ. Their laws help to ensure that the transformations do not change the program's behavior, when the provisos (preconditions) they state hold. Our work relies on their ideas, and we derived some refactorings for AspectJ using their laws. However, their laws are bi-directional, whereas our laws use uni-directional laws that are oriented to improve code quality.

Iwamoto and Zhao [Iwamoto and Zhao 2003], just as our work, take into account aspect-oriented refactorings. But, their refactorings are concerned with restructuring Java programs to AspectJ (refactoring OO to AO programs), whereas our work is related to refactor AspectJ constructs (improving AspectJ programs). As with our work, they present a collection of aspect-oriented refactorings, but most of them are aspect-aware OO refactorings (also related to OO programs).

Another related work is Hannemann et al. [Hannemann et al. 2005]. Like our work, they propose a set of aspect-oriented refactorings. Their refactorings are grouped by three distinct categories as mentioned in Section 3. They use testing to check that refactorings do not change the behavior of programs, whereas we are concerned with (static) proofs of correctness for refactorings.

6. Conclusions

In this paper, we have presented programming laws for aspect-oriented programming and used them to define behavior-preserving transformations for AspectJ constructs. The laws are simple and localized, which should make it easy to prove their soundness. Moreover, we also use a comprehensive set of aspect-oriented programming laws, already proved to be sound, from the literature. Those laws help us to derive the refactoring transformations that we use in optimization.

As future work, we plan to augment our set of laws to handle more AspectJ constructs. Moreover, we also intend to use those set of laws to derive new refactorings and to derive those already proposed in the literature. Another interesting issue is about soundness. The new laws we proposed do not yet have a formal soundness proof. We plan to fix this limitation in future work. Currently, we are also conducting more case studies to evaluate our proposed laws and refactorings.

Our main contribution is that we have shown how to use the proposed laws and refactorings to optimize compilation of JML in our compiler, ajmlc. To better explain the impacts of such optimizations, we have conducted a case study on four Java programs. The results provided evidence that the ajmlc compiler produces smaller source and bytecode instrumentation when it employs the transformations proposed by this work. We also considered the two existing AspectJ weavers (ajc and abc) that ajmlc supports. The case study showed that the instrumented bytecode produced by the optimizing ajmlc compiler is much faster when using the abc weaver. Such results are essential when considering constrained environments such as Java ME. To the best of our knowledge, this is the first work that concerns assertion checking code optimization.

Although we use the laws and refactorings presented here for optimization, they are of more general utility. As a result, besides their use in optimizing JML compilers, one could apply these transformations to other AspectJ programs.

Acknowledgements

This work was partially supported by Brazilian research agency FACEPE. The work of Leavens was partially supported by a US National Science Foundation grant, CNS 08-08913. Special thanks to Paulo Borba for discussions about issues of this paper.

References

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

Briand, L. C., Dzidek, W. J., and Labiche, Y. (2005). Instrumenting Contracts with Aspect-Oriented Programming to Increase Observability and Support Debugging. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 687–690, Washington, DC, USA. IEEE Computer Society.

Burdy, L., Cheon, Y., Cok, D. R., Ernst, M. D., Kiniry, J. R., Leavens, G. T., Leino, K. R. M., and Poll, E. (2005). An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer (STTT)*, 7(3):212–232.

Cheon, Y. (2003). *A runtime assertion checker for the Java Modeling Language*. Technical report 03-09, Iowa State University, Department of Computer Science, Ames, IA. The author's Ph.D. dissertation.

Cole, L. and Borba, P. (2005). Deriving refactorings for aspectj. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 123–134, New York, NY, USA. ACM.

Cole, L., Borba, P., and Mota, A. (2005). Proving aspect-oriented programming laws. In Leavens, G. T., Clifton, C., and Lämmel, R., editors, *Foundations of Aspect-Oriented Languages*.

Feldman, Y. A., Barzilay, O., and Tyszberowicz, S. (2006). Jose: Aspects for design by contract80-89. *sefm*, 0:80–89.

Fowler, M. et al. (1999). *Refactoring: improving the design of existing code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Hannemann, J., Murphy, G. C., and Kiczales, G. (2005). Role-based refactoring of crosscutting concerns. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 135–146, New York, NY, USA. ACM.

Hoare, C. A. R., Hayes, I. J., Jifeng, H., Morgan, C. C., Roscoe, A. W., Sanders, J. W., Sorensen, I. H., Spivey, J. M., and Sufrin, B. A. (1987). Laws of programming. *Commun. ACM*, 30(8):672–686.

Iwamoto, M. and Zhao, J. (2003). Refactoring aspect-oriented programs. In Akkawi, F., Aldawud, O., Booch, G., Clarke, S., Gray, J., Harrison, B., Kandé, M., Stein, D., Tarr, P., and Zakaria, A., editors, *The 4th AOSD Modeling With UML Workshop*.

Kiczales, G. (1996). Aspect-oriented programming. *ACM Comput. Surv.*, page 154.

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G. (2001). An Overview of AspectJ. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353, London, UK. Springer-Verlag.

Laddad, R. (2003). *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications Co., Greenwich, CT, USA.

Laddad, R. (2006). *Aspect Oriented Refactoring*. Addison-Wesley Professional.

Leavens, G. T. (2006). JML's rich, inherited specifications for behavioral subtypes. In Liu, Z. and Jifeng, H., editors, *Formal Methods and Software Engineering: 8th International Conference on Formal Engineering Methods (ICFEM)*, volume 4260 of *Lecture Notes in Computer Science*, pages 2–34, New York, NY. Springer-Verlag.

Leavens, G. T., Baker, A. L., and Ruby, C. (2006). Preliminary design of JML: a behavioral interface specification language for java. *SIGSOFT Softw. Eng. Notes*, 31(3):1–38.

Monteiro, M. P. and Fernandes, Jo a. M. (2005). Towards a catalog of aspect-oriented refactorings. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 111–122, New York, NY, USA. ACM.

Opdyke, W. F. (1992). *Refactoring object-oriented frameworks*. PhD thesis, Champaign, IL, USA.

Rebêlo, H., Lima, R., Cornélio, M., Leavens, G. T., Mota, A., and Oliveira, C. (2009). Optimizing JML feature compilation in ajmlc using aspect-oriented refactorings. Technical Report CS-TR-09-05, 4000 Central Florida Blvd., Orlando, Florida, 32816-2362.

Rebêlo, H., Soares, S., Lima, R., Borba, P., and Cornélio, M. (2008a). JML and aspects: The beneifts of instrumenting JML features with AspectJ. In *Seventh International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2008)*, number CS-TR-08-07 in Technical Report, pages 11–18, 4000 Central Florida Blvd., Orlando, Florida, 32816-2362. School of EECS, UCF.

Rebêlo, H., Soares, S., Lima, R., Ferreira, L., and Cornélio, M. (2008b). Implementing java modeling language contracts with aspectj. In *SAC '08: Proceedings of the 2008 ACM symposium on Applied computing*, pages 228–233, New York, NY, USA. ACM.

Roberts, D. B. (1999). *Practical analysis for refactoring*. PhD thesis, Champaign, IL, USA. Adviser-Johnson, Ralph.

Sampaio, A. (1997). *An Algebraic Approach to Compiler Design*. World Scientific.

Wampler, D. (2006). Contract4J for Design by Contract in Java: Design Pattern-Like Protocols and Aspect Interfaces. In *ACP4IS Workshop at AOSD 2006*, pages 27–30.

A. Online Appendix

We invite researchers to replicate our case study. Annotated source code with JML and our ajmlc compilers (the non-optimized and optimized version), AspectJ weavers (ajc and abc), JML classical compiler (jmlc), and our results are available at:
`http://www.cin.ufpe.br/~hemr/JMLAOP/sblp09`.