Optimizing JML Features Compilation in Ajmlc Using Aspect-Oriented Refactorings

Henrique Rebêlo, Ricardo Lima, Márcio Cornélio,
Gary T. Leavens, Alexandre Mota, César Oliveira

CS-TR-09-05
April 2009

Keywords: ajmlc, runtime assertion checking, optimization, refactoring, semantics preservation, laws of programming, formal methods, formal interface specification, programming by contract, aspect-oriented programming, JML language, Java language, AspectJ language

2009 CR Categories: D.2.1 [*Software Engineering*] Requirements/ Specifications — languages, tools, JML; D.2.2 [*Software Engineering*] Design Tools and Techniques — computer-aided software engineering (CASE); D.2.4 [*Software Engineering*] Software/Program Verification — Assertion checkers, class invariants, formal methods, programming by contract, reliability, tools, validation, JML; D.2.5 [*Software Engineering*] Testing and Debugging — Debugging aids, design, testing tools, theory; F.3.1 [*Logics and Meanings of Programs*] Specifying and Verifying and Reasoning about Programs — Assertions, invariants, pre- and post-conditions, specification techniques.

This is a preprint of a paper that will appear in the proceedings of the XIII Brazilian Symposium on Programming Languages (SBLP 2009), Gramado-RS, Brazil, August 19-21.

School of Electrical Engineering and Computer Science
University of Central Florida
4000 Central Florida Blvd.
Orlando, FL 32816-2362 USA

Optimizing JML Features 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 small bytecode size and running time, there is still a need for optimization of bytecode size and running time. To do this optimization while preserving the semantics of the resulting code, we optimize using 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 object-oriented program is a useful activity known as refactoring [Opdyke 1992, Roberts 1999, Fowler et al. 1999]. By changing attributes and methods between classes or splitting a complex class into several classes while preserving observable behavior, we can improve certain code's modularity, decrease its size, and so on. However, if these refactoring transformations are to be applied automatically by a compiler, the consequences of mistakes are greatly amplified. Hence it is important that such refactorings be very trustworthy, that is, that they correctly preserve the program's observable behavior.

Our approch to solving this problem is to design refactorings using programming laws [Hoare et al. 1987, Cornélio 2004, Borba et al. 2004],

and thus the refactorings are provably correct [Sampaio 1997]. Our compiler, ajmlc, generates aspect-oriented code [Kiczales 1996], written using AspectJ [Kiczales et al. 2001], which is 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 use 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 these laws and other proposed to derive optimizing transformations, which are refactoring rules applied in a particular direction. We use Cole and Borba's laws, which they have already proven correct [Cole et al. 2005], to prove the correctness of some refactoring transformations.

The ajmlc compiler was described in our previous work by Rebelo [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 can also be applied to constrained environments such as Java ME applications. While there are several related work 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. Such optimization 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 two systems. Third, to the best of our knowledge, this is the first work that shows how to optimize asserting 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 two systems. In Section 5, we discuss related work and in Section 6, we present our conclusions.

2. An Overview of JML

The Java programming language has assertions, but otherwise has no built-in support for Design by Contract (DbC). The Java Modeling Language

```
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.

(JML) [Leavens et al. 2006, Leavens 2006] is a behavioral interface specification language for Java that provides DbC support for Java.

JML includes a number of constructs to declaratively specify runtime behavior. Classes are specified by specifying their fields, invariants over those fields, and by specifying constructors and methods. (In the following, we refer to both constructors and methods as "methods" when there is no need to distinguish them.) Methods specifications are composed of pre- and postconditions. All these JML specifications are written in Java code files using special comments. Figure 1 illustrates a simple JML specification concerning the contract for a single method `div`. The contract is composed of a precondition, requiring `b > 0` and a postcondition, ensuring that the methods result is `a / b`.

The benefits of adding JML annotations for a Java source code include the following: (1) precise description of what the code should do; (2) efficient discovery and correction of bugs; (3) early discovery of incorrect client usage of classes; (4) reduced chance of introducing bugs as the application evolves, and (5) precise documentation that is always in accordance with application code.

There are a number of tools that work with JML [Burdy et al. 2005], including ESC/Java2 [Flanagan et al. 2002, Cok and Kiniry 2005] and 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 (details were omitted for simplicity).

3. Laws and Refactorings

For establishing a systematic and rigorous basis for programming refactorings, algebraic laws of programming [Hoare et al. 1987] can be used. Moreover, we can use laws to design correct compilers and code optimizers [Sampaio 1997].

```
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.

Optimization is our focus in this paper. To illustrate the use of the algebraic approach in imperative programs, consider two laws in an imperative language [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 the value of a variable 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 $\langle$*void assignment*$\rangle$

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

Law $\langle$*unit-skip*$\rangle$

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

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

3.1. Aspect-Oriented Refactoring

Besides object-oriented programming (OO), refactorings are also quite useful for restructuring aspect-oriented programming (AO) elements, increasing legibility,

extensibility, maintainability, and performance. However, aspect-oriented refactorings, in contrast with traditional OO refactorings, involve AO constructs, such as aspects and advice. Hannemann [Hannemann et al. 2005] classifies aspect-oriented refactorings into three distinct groups:

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

In this classification, our 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. By observing that problem, Cole and Borba describe a set of AspectJ programming laws that give us a basis for proving that the transformations preserve behaviour and, therefore, are indeed refactorings [Cole and Borba 2005]. Thus, some refactorings we present are derived from, and proven correct using [Cole et al. 2005], the AspectJ programming laws proposed by Cole and Borba. Other refactorings are derived from laws also proposed in this work. Soundness of our laws using formal semantics will be treated in future work.

Notation

The subset of laws and refactorings we describe are written using two boxes written side by side, along with a **provided** clause. This clause give conditions, also known as provisos, all of which must be true for the the law or refactoring to be correctly applied. The notation "$(\rightarrow)$" 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 **paspect** to denote a privileged aspect declaration for simplicity. We easily derive this law by applying Law 2 $\langle make\ aspect\ privileged \rangle$ and $\langle add\ empty\ aspect \rangle$ from Cole and Borba's laws [Cole and Borba 2005]. Both laws are applied from right-to-left.

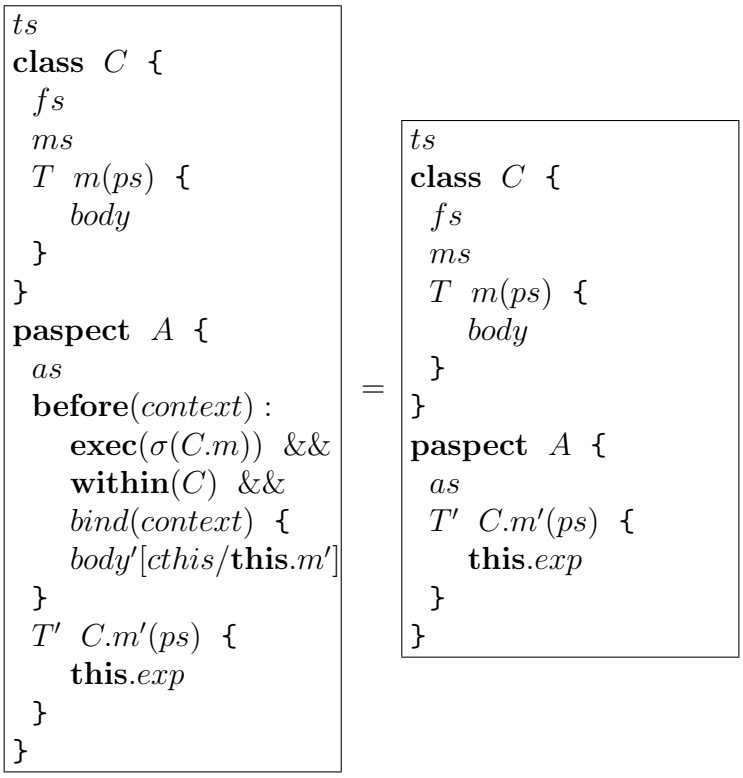
Law 1. $\langle$*remove empty privileged aspect*$\rangle$

$$\boxed{\begin{array}{l} ts \\ \textbf{paspect}\ \ A\ \ \texttt{\{} \\ \texttt{\}} \end{array}} = \boxed{ts}$$

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

Law 1 is useful in ajmlc optimization when no instrumentation code is provided, resulting in an empty privileged aspect. Ajmlc only generates an empty aspect when no JML annotations are provided or when an empty class is being compiled. It is important to emphasize that the classical JML compiler (jmlc) [Cheon 2003] always generates 11.0 KB (source code instrumentation) and 5.93 KB (bytecode instrumentation) for empty classes [Rebêlo et al. 2008a].

Law 5. ⟨*remove before-execution*⟩



provided
(→) **before** advice does not contribute to execution flow of the affected join point $\sigma(C.m)$, or type C is declared **abstract** or it is declared as an **interface**;
(→) the designator **within** appears in the **before** advice. □

The current law (**Law** 5) shows a transformation which removes the **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 bind the AspectJ advice parameters (`this`, `args`). Additionally, we use the AspectJ designator $\mathbf{within}(C)$ to prevent the **before** advice to affect executions of method m in subtypes of C. An important expression is $body'[cthis/\mathbf{this}.member\_reference]$, which substitutes $cthis$ for all occurrences of **this** within $body'$; a $member\_reference$ could denote C's fields, or

Table 1. Summary of Aspect-Oriented Laws and Refactorings

Laws	Refactorings
1. remove empty privileged aspect	1. inline method intertype within advice
2. move advice body to other advice	2. merge distinct advice
3. replace method intertype reference with method intertype implementation within advice	3. split around-execution into after-execution returning and after-execution throwing
4. remove method intertype related to advice	4. extract aspect method
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	

methods denoted by fs and ms respectively (and also including method intertypes, such as m').

The first proviso states that the **before** advice does not add any behavior to the affected method m. Thus, we can remove it. Moreover, we also can remove the **before** advice if the declared type is **abstract** or if it is an **interface**. This condition is always valid when the second proviso holds, because the **within** designator constrains the advice's application in subtypes and we cannot instantiate a concrete class when we have an **abstract** or an **interface** type. Therefore, we always can remove such advice.

This is the simplest law to remove advice, thus it can be applied to other advice as well (see Table 1). 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 provisos must consider different situations, even though the main result is the same advice elimination.

In the context of JML and ajmlc, such a law 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 **`true`**, we can remove the related advice (see the first proviso).

The next rule is a refactoring (**Refactoring**) 1, in that, when applied from left-to-right, inlines the method intertype implementation within **before** advice. This transformation is useful because the method intertype is only referenced by one advice so that we can remove the method intertype and migrate its implementation within the advice. The derivation of this refactoring involves two other simple laws. 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 **before** advice with its implementation, and (2) apply **Law** 4 $\langle remove\ method\ intertype\ related\ to\ advice \rangle$, removing the method intertype m'. We do not shown their equations, because they are simple.

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]. Each of those laws and refactoring are summarized in Table 1.

Refactoring 1 is useful to ajmlc's optimization when, for example, a **before** advice is checking a precondition, and this advice references a method intertype with the precondition predicate which 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 generated code by ajmlc. The result provided by applying such a refactoring is shown in Figure 3.

```
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.

Refactoring 1. ⟨*inline method intertype within before-execution*⟩

$$
\begin{array}{l}
ts \\
\textbf{class}\ C\ \{ \\
\quad fs \\
\quad ms \\
\quad T\ \ m(ps)\ \{ \\
\quad\quad body \\
\quad \} \\
\} \\
\textbf{paspect}\ A\ \{ \\
\quad as \\
\quad \textbf{before}(context) : \\
\quad\quad \textbf{exec}(\sigma(C.m))\ \&\& \\
\quad\quad \textbf{within}(C)\ \&\& \\
\quad\quad bind(context)\ \{ \\
\quad\quad body'[cthis/\textbf{this}.m'] \\
\quad \} \\
\quad T'\ \ C.m'(ps)\ \{ \\
\quad\quad \textbf{this}.exp \\
\quad \} \\
\}
\end{array}
\quad = \quad
\begin{array}{l}
ts \\
\textbf{class}\ C\ \{ \\
\quad fs \\
\quad ms \\
\quad T\ \ m(ps)\ \{ \\
\quad\quad body \\
\quad \} \\
\} \\
\textbf{paspect}\ A\ \{ \\
\quad as \\
\quad \textbf{before}(context) : \\
\quad\quad \textbf{exec}(\sigma(C.m))\ \&\& \\
\quad\quad \textbf{within}(C)\ \&\& \\
\quad\quad bind(context)\ \{ \\
\quad\quad body'[cthis/\textbf{this}.exp] \\
\quad \} \\
\}
\end{array}
$$

provided
(→) exp does not appear in **before** advice or as;
(→) m is not referenced from C, ts, or as. □

Other laws and refactorings

Besides the presented laws and refactorings, as shown in Table 1, we derived others with great results in the context of ajmlc optimization. For example, the **Law** 2 $\langle move\ advice\ body\ to\ other\ advice \rangle$, enables one, for instance, to move the implementation of a **`before`** advice to the body of an **`around`** advice (before the call to **`proceed`**). The only precondition is that both advice have to affect the same method on a specific type. By using **Law** 2 and **Law** 5, we derive the the **Refactoring** 2 $\langle merge\ distinct\ advice \rangle$, which enables one, for example, to merge a **`before`** advice into an **`around`** advice. After **Law** 2, we apply **Law** 5 to remove the empty **`before`** advice.

In the context of JML and ajmlc optimization, when we have a scenario with a static method affect by one **`before`** advice to check preconditions and one **`around`** advice to check postconditions. Thus, we can merge such advice by means of **Refactoring** 2. This happens, because the method is static and its advice do not affect subtypes.

Soundness

As we mentioned before, programming laws [Hoare et al. 1987] define equivalence between two programs, given that some conditions are respected. However, the proof of the behaviour preserving property of programming laws is not trivially demonstrated. So, we follow the ideas of Cole and Borba's laws [Cole and Borba 2005]. As some of laws we propose are composition of their laws, we can assume that our laws are proven correct. This happens because they show how to prove that an aspect-oriented programming law preserves behaviour using a formal semantics [Cole et al. 2005].

However, there are laws in this work that are not derived from Cole and Borba's work [Cole and Borba 2005]. Those soundness using a formal semantics is a desirable property, thus, as future work we intend to use the same formal semantics to prove that these laws are behaviour-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, involve only local changes, and each one concerns only a specific AspectJ construct.

Table 2. Laws and Refactorings in the JAccounting and Bomber systems

JAccounting		Bomber	
	Qty		Qty
Law 1	28	Law 1	5
Law 2	8	Law 2	3
Law 3	2	Law 3	2
Law 4	5	Law 4	2
Law 5	33	Law 5	10
Law 6	30	Law 9	2
Law 9	34	Law 10	2
Law 10	34	Refactoring 1	2
Refactoring 1	2	Refactoring 2	3
Refactoring 2	8	Refactoring 3	20
Refactoring 3	11		

4. Case Study

This section presents the results of a case study involving the JAccounting [1] and Bomber [2] systems. We apply our proposed laws and refactorings to analyze their benefits brought for both systems. The quantification of the applied set of laws and refactorings are summarized in Table 2.

We have compiled these two systems, after annotating them with JML, using both jmlc [Cheon 2003] and ajmlc [Rebêlo et al. 2008b]. For ajmlc we employed two versions: with and without the refactorings (optimizations) described above. Moreover, we used ajmlc with two different weaving processes: the standard AspectJ compiler (ajc), and abc [Avgustinov et al. 2005]. The difference is that abc itself includes various optimizations.

Our case study considers a Java ME application because ajmlc, unlike jmlc, can 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. Some features of Bomber include explosions, war tanks, planes sounds, clouds, etc.

[1] https://jaccounting.dev.java.net.
[2] http://j2mebomber.sourceforge.net.

Code size and performance statistics

Throughout our assessment of the proposed laws and refactorings, we gathered some measurements that demonstrated an improvement in both code size and performance of the optimized ajmlc aspects code. Table 3 and Table 4 present the results that we obtained by assessing the proposed refactorings. As observed, we analyzed instrumented source code size (denoted by ISC in Table 3) and instrumented bytecode size, all in megabytes (MB). Moreover, we also analyzed running time measured in milliseconds (msec).

In relation to code size, we observed that the optimized ajmlc aspect code is smaller than the non-optimized code, both in instrumented source code and bytecode. It is worth nothing that this effect on the bytecode is enhanced, producing far smaller class files, when the abc weaver is employed (see Table 3). It is also clear that ajmlc always 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 in a optimized way and using the abc weaver. This indicates that the compilation techniques for aspect oriented programs are still in a stage of continuous evolution.

Concerning running time, we observed that the optimized ajmlc aspects code executes faster than the non-optimized one (see Table 4). As also shown in Table 4, the running time is far faster when the optimized ajmlc employs abc weaver. Additionally, as noted, the running time of the ajmlc aspects code is always faster than the jmlc code, even in a non-optimized way. Such Bad performance in jmlc is due to many reflective calls in the jmlc generated code. Its important to note that we only measured the execution time of methods compiled with jmlc in the JAccounting system. 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 with Java ME API.

5. Related Work

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

The classical 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

Table 3. Code size measurements

Application	Original code (MB)	Optimized (MB)	Decrease (%)
JAccounting			
ajmlc/ISC	1.28	1.07	16.40
ajmlc/ajc	3.51	2.20	37.32
ajmlc/abc	2.83	0.85	69.96
jmlc/ISC	5.30	-	-
jmlc	1.95	-	-
Bomber			
ajmlc/ISC	0.95	0.83	12.63
ajmlc/ajc	2.26	1.47	34.95
ajmlc/abc	0.86	0.61	29.06
jmlc/ISC	4.53	-	-
jmlc	1.60	-	-

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 as 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.

Table 4. Running time measurements

Method	Original (msec)	Optimized (msec)	Decrease (%)
ajmlc/ajc			
JAccounting/getCreated	0.05915	0.03548	40.01
JAccounting/getCompanyKey	0.05636	0.03038	46.09
JAccounting/perform2	5.78228	4.97493	13.96
Bomber/handle	3.47174	2.96776	14.51
Bomber/getRadius	3.97096	3.14669	20.75
Bomber/getDamage	3.52824	3.32367	5.79
ajmlc/abc			
JAccounting/getCreated	0.05042	0.03289	34.76
JAccounting/getCompanyKey	0.05105	0.02891	43.36
JAccounting/perform2	5.74734	4.90301	14.69
Bomber/handle	0.06690	0.03841	42.58
Bomber/getRadius	0.06348	0.03715	41.47
Bomber/getDamage	0.05259	0.03080	41.43
jmlc			
JAccounting/getCreated	0.32706	-	-
JAccounting/getCompanyKey	0.33461	-	-
JAccounting/perform2	6.91035	-	-

6. Conclusions

In this paper, we have presented a set of programming laws for aspect-oriented programming and used them to define behaviour-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 proven sound, from the literature. Those laws help us to derive our refactoring transformations that we use in optimization.

As future work, we intend to augment our set of laws to handle more AspectJ constructs. Moreover, we also intend to use those set of laws to derive more new refactorings or to derive ones proposed in the literature. Another interesting issue is about soundness. The new laws we proposed do not yet have a formal semantics and soundness proof. This is a limitation of these new laws, that we intend to fix in future work.

Our main contribution is that we show 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 conducted a case study. We analyzed two Java systems, one of which is a Java ME application. The results we obtain, provide evidence that the ajmlc compiler produces smaller source and bytecode instrumentation when it employs the transformations proposed by this work. We consider the two existing AspectJ weavers (ajc and abc) that ajmlc supports. Additionally, we also conducted a performance comparison using the case study. We obtain results, which indicate us that the instrumented bytecode produced by ajmlc compiler, when using the proposed transformations, is much faster than one without optimizations. As we note, the better results involve the ajmlc compiler using the abc weaver. Such results are essential when considering constrained environments such as Java ME.

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 can apply these transformations to other AspectJ programs.

Acknowledgements

This work was partially supported by Brazilian research agency FACEPE and by grants from the US National Science Foundation numbered 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.

Borba, P., Sampaio, A., Cavalcanti, A., and Cornélio, M. (2004). Algebraic reasoning for object-oriented programming. *Sci. Comput. Program.*, 52(1-3):53–100.

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.

Cok, D. R. and Kiniry, J. R. (2005). ESC/Java2: Uniting ESC/Java and JML: Progress and issues in building and using ESC/Java2, including a case study involving the use of the tool to verify portions of an Internet voting tally system. In Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., and Muntean, T., editors, *Construction and Analysis of Safe, Secure, and Interoperable Smart devices (CASSIS 2004)*, volume 3362 of *Lecture Notes in Computer Science*, pages 108–128. Springer-Verlag.

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*.

Cornélio, M. L. (2004). *Refactoring as Formal Refinements*. PhD thesis.

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

Flanagan, C., Leino, K. R. M., Lillibridge, M., Nelson, G., Saxe, J. B., and Stata, R. (2002). Extended static checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI'02)*, volume 37(5) of *SIGPLAN*, pages 234–245, New York, NY. ACM.

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., 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.