Optimizing Generated Aspect-Oriented Assertion Checking Code for JML Using Programming Laws: An Empirical Study

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

CS-TR-10-01
February, 2010

Keywords: Aspect-oriented programming, refactoring, programming laws, 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.

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

Optimizing Generated Aspect-Oriented Assertion Checking Code for JML Using Programming Laws: An Empirical Study[☆]

Henrique Rebêlo[a,*], Ricardo Lima[a], Gary T. Leavens[b], Márcio Cornélio[c], Alexandre Mota[a], César Oliveira[a]

[a]*Informatics Center — Federal University of Pernambuco*
Caixa Postal 7851, 50740-540 — Recife — PE — Brazil
[b]*School of Electrical Engineering and Computer Science — University of Central Florida*
4000 Central Florida Blvd. — Orlando — FL — USA
[c]*Departamento de Estatística e Informática — Universidade Federal Rural de Pernambuco*
Rua Dom Manuel de Medeiros, s/n 52171-900 — Recife — PE — Brazil

Abstract

Aspect-oriented programming (AOP) enables the effective modularization of well-known crosscutting concerns. To take advantage of AOP, there are many techniques, including AOP laws, for a systematic refactoring of crosscutting concerns to aspects. However, there is also a need for supporting the systematic refactoring of AOP constructs. Existing techniques for aspect-oriented refactoring are too coarse-grained and make it too difficult to assure that the transformations preserve behavior and are indeed refactorings. This paper presents a catalogue of AOP laws towards a systematic refactoring of AOP constructs. As long as these laws are fine-grained, they make it easier to verify that the transformations they provide preserve behavior. Furthermore, as these laws can provide space and time optimization, we conduce an empirical study using four applications in optimized and non-optimized versions generated by ajmlc, a new JML compiler, presented in a previous work, which generates aspects that enforce JML contracts during runtime. We show that the AOP laws provide a significant improvement, regarding bytecode size and running time, in the aspect code generated by an optimized version of the ajmlc compiler.

Keywords: Aspect-oriented programming, Refactoring, Programming laws, JML

1. Introduction

Aspect-oriented programming (AOP) [2] is a well-known technique that explicitly supports the modularization of crosscutting concerns. With the emergence of AOP, many techniques has been proposed to help programmers in applying refactorings on scattered and tangled implementations of crosscutting concerns to aspects (OO-to-AO refactorings) [3, 4, 5]. Refactoring [6, 7, 8] is an useful technique, which appeared in the context of object-oriented programming (OOP), for improving code structure through behavior-preserving program transformations. In addition to refactoring of crosscutting concerns, the importance of supporting aspect-oriented refactorings (AO-to-AO) has been recongnized [9, 10]. Aspect-oriented (AO) refactorings are employed for improving existing AO code. Through refactoring, one might increase code modularity, reusability, decrease code size, etc. However, transformations performed through refactorings might be difficult to understand. It is particularly true when they involve global changes. Furthermore, as refactorings are usually coarse-grained (involve several language constructs), it might be hard to ensure that the transformations preserve program behavior and are indeed refactorings.

Our approach for solving this problem is to derive aspect-oriented refactorings by using programming laws [11]. The strategy establishes constraints on the program behavior. Two programs are said to be equivalent if they respect such constraints. Thus, the effectiveness of programming laws is to show that some particular aspect-oriented transformations are indeed refactorings. In contrast with refactorings, laws are simpler, involve only local changes, and are fine-grained (each law focus on a specific language construct).

Programming laws have been successfully defined for the most important programming models [11, 12, 13, 14, 15]. More recently, laws for aspect-oriented programming [5] were also introduced to derive aspect-oriented refactorings and show that they are behavior preserving transformations. However, such aspect-oriented programming laws only provide support for refactorings of crosscutting concerns (OO-to-AO refactorings). In this paper, we focus on this problem and introduce aspect-oriented programming laws that are useful to restructure aspect-oriented constructs (AO-to-AO refactorings).

Another goal of this work is to apply aspect-oriented transformations to automatically optimize aspect-oriented programs. The consequences of incorrect transformations can be greatly amplified when such an automatic approach is applied. We use the rules defined through the aspect-oriented programming laws

[☆]This paper is an extension of [1].
[*]Corresponding author.
Email addresses: `hemr@cin.ufpe.br` (Henrique Rebêlo), `rmfl@cin.ufpe.br` (Ricardo Lima), `leavens@eecs.ucf.edu` (Gary T. Leavens), `marcio@dsc.upe.br` (Márcio Cornélio), `acm@cin.ufpe.br` (Alexandre Mota), `calo@cin.ufpe.br` (César Oliveira)

to provide trustworthy transformations.

We enriched the ajmlc compiler [16] with the aspect-oriented transformations proposed in this paper. Ajmlc takes an input code written in the Java Modeling Language (JML) [17, 18] and generates AspectJ aspects to check the JML specifications at runtime. Unlike the classical JML compiler, jmlc [19], ajmlc does not use Java's reflection facilities, and thus can also be applied to constrained environments such as Java ME applications. In order to optimize the generated AspectJ code, we draw on the work of Cole and Borba [5]. Their laws establish how to restructure AspectJ code, by adding or removing AspectJ constructs. We use their laws and have proposed others to derive refactoring rules that deal specifically with optimization. Soundness of a few of the refactoring transformations follows from the soundness of Cole and Borba's laws, which have been proven previously [20].

The recognition that the Design by Contract (DbC) [21] technique can be implemented and better modularized using AOP is not new. There are several related works that implement such dynamic contract checking using aspects [22, 23, 24], but none of them optimizes the generated aspects. The 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 an empirical study with four applications. Third, to the best of our knowledge, this is the first work that shows how to optimize aspect-oriented assertion checking code. While we present almost of 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. We discuss related research in Section 3. After that, in Sections 4 and 5, we present the proposed aspect-oriented laws and refactorings, respectively. These laws and refactorings are discussed in Section 6. In Section 7, we quantitatively and qualitatively assess the impact of the proposed laws and refactorings in a case study involving four applications. Finally, in Section 8, we present our conclusions.

2. An Overview of JML

Java has assertions, but no other built-in support for DbC [21]. The Java Modeling Language (JML) [25, 18] 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`.

```
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.

There are a number of tools that work with JML [17], including the classical JML compiler (jmlc) [19]. Like jmlc, our ajmlc compiler [16] 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).

3. Literature Review on Aspect-Oriented Refactoring

Aspect-oriented refactorings contrast with classical object-oriented refactorings in that they involve AOP constructs. Hannemann, Murphy, and Kiczales [4] classify aspect-oriented refactorings into three categories: (i) aspect-aware OO refactorings, (ii) aspect-oriented refactorings, and (iii) refactorings of crosscutting concerns. In this section, we present a review of each category of aspect-oriented refactorings. We also briefly discuss some works in the literature that are related to these categories. Its important to note that once the reader is aware of those categories of aspect-oriented refactorings, the rest of this paper only focuses on the second category.

3.1. Aspect-aware OO Refactoring

Several authors [26, 9, 27] have identified limitations when applying conventional OO refactorings [8] in the presence of aspects. The main limitation is that such OO refactorings modify the structure of join points of the base program [1]. To handle this problem, Hanenberg, Oberschulte, and Unland (HOU) [26]

[1]Base program is related to OO code of an AO system.

draw on the work of Opdyke [6] by using preconditions which have to be respected when applying a conventional refactoring in the presence of aspects. They introduce aspect-aware versions of traditional OO refactorings, such as *rename method* [8, p. 273], in order to exemplify the preconditions which make the refactoring aspect-aware.

As with HOU [26], Iwamoto and Zhao [9] demonstrated modifications to OO refactorings in order to make them aspect-aware. However, instead of using preconditions, Iwamoto and Zhao proposed guidelines joined to modifications of OO refactorings to make them aspect-aware. In addition, they also discuss implementation issues of their tool that supports automatic refactoring of AspectJ programs. Similarly, Wloka [27] discusses OO refactorings in the presence of aspects. Nonetheless, that author does not tackle the issue on how to make OO refactorings aspect-aware.

3.2. Aspect-Oriented Refactorings

Aspect-oriented refactorings are the activity to restructure AO constructs of an AO system. To achieve that, we need new refactorings that, in addition to the object-oriented ones, restructure the aspect code (i.e., advice and pointcuts). In this context, Iwamoto and Zhao [9] also proposed new refactorings, besides aspect-aware refactorings discussed earlier, for restructuring AOP constructs. They show examples and also give guidelines on how to apply such AO refactorings they propose. The authors summarized their new refactorings for AOP constructs as a catalogue which can help AO developers during refactoring process. Similarly, Monteiro and Fernandes [28, 10] proposed a catalogue of AO refactorings and described them in a similar way to Fowler's object-oriented ones [8]. For example, Monteiro and Fernandes, demonstrated several AO refactorings by means of illustrative examples, such as *move field from class to intertype*, *move method from class to intertype*, and among others.

All the aforementioned aspect-oriented refactorings are fine-grained in relation to crosscutting concern modularization, so that they do not allow the designer to reason about the elements involved neither in a specific nor groups of crosscutting concerns. Therefore, coarse-grained refactorings for modularizing crosscutting concerns have emerged as outlined in the next section.

Its important to note that refactorings involve two kinds of granularity depending on the point of view. For example, by considering a refactoring itself, it can be coarse-grained. As discussed in the introduction (Section 1), a refactoring is coarse-grained whether it involves several programming constructs. However, a refactoring can also be fine-grained in a crosscutting concern modularization view. Just one refactoring is usually not enough to provide a complete modularization of a particular crosscutting concern. Thus, we can reason that a single refactoring is fine-grained in relation to a complete crosscutting concern modularization.

Also considering fine-grained refactorings, this paper proposes a catalogue of aspect-oriented programming laws (Table 1) useful for deriving AO refactorings (see Section 4.1 for more details). The refactorings we derive are fine-grained in relation to crosscutting concern modularization and coarse-grained because they affect more than one programming construct at a time. On the other hand, the laws we propose to derive AO refactorings are fine-grained in both perspectives. An aspect-oriented programming law is much simpler than a refactoring; it involves only one aspect-oriented programming construct at a time.

3.3. Refactorings of Crosscutting Concerns

Besides restructuring OO code, refactorings are useful to migrate scattered and tangled implementations of crosscutting concerns into aspects (OO-to-AO refactorings). Several approaches [3, 4, 5, 29] have been proposed on how to modularize/refactor the so-called crosscutting concerns, which harm code quality, into aspects. In addition, we have in the literature, cookbooks to guide the "aspectization" of specific crosscutting concerns, such as exception handling [30, 31], distribution and persistence [32], and concurrency control [33]. We also have works that show how to modularize design patterns with aspects [34, 35]. By using a different strategy, Binkley *et al.* [3] present a human-guided automated approach that support the composition of refactorings which extract Java code fragments into aspects. Their approach are fine-grained, so that their approach is restricted to the replacement of Java code fragments with pointcuts and advice.

Hannemann, Murphy, and Kiczales (HMK) [4] propose role-based refactoring in that crosscutting concerns are described in terms of abstract roles. The underlying idea is that refactoring instructions for crosscutting concerns are written in terms of those roles. In this way, a developer to apply a refactoring needs to map a subset of such roles to concrete program elements. Tool support is provided for developers in order to refactor the code elements implementing the concern. As with HMK's approach, Marin *et al.* [29] proposed refactorings based on crosscutting concern types. A concern type consists of a general intent, an implementation idiom, and one aspect language mechanism to address it. Similarly to HMK's work, Marin *et al.* also individualize and describe groups of crosscutting concerns sharing common properties. However, while the former rely on abstract roles, the latter restricts the concern classification to implementation idioms or specific AOP mechanisms. Furthermore, both HMK and Marin *et al.* approaches with concern groups with similar structures are coarse-grained.

Cole and Borba [5] introduce a set of thirty aspect-oriented programming laws for deriving 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. The transformations they propose are useful elements in the migration of crosscutting concerns to aspects; even though, they do not address crosscutting concerns directly. Thus, their AspectJ laws are fine-grained transformations. They describe code transformations with their respective preconditions that can occur as steps in the (derived) aspect refactoring of a crosscutting concern, but not the concern itself (as HMK [4] and Marin *et al.* [29] approaches). In addition, their laws are bi-directional, this implies that they do not always increase code quality.

4. Laws

For establishing a systematic and rigorous basis for optimization via program transformation, we use algebraic laws of programming [11] to design code optimizers [36]. We illustrate the use of the algebraic approach by considering two programming laws [11]: one related to the assignment command, and the other 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*.

Laws ⟨*void assignment*⟩ and ⟨*unit-skip*⟩

$$(x := x) = \textbf{skip}$$

$$(\textbf{skip}; \mathit{stmt}) = (\mathit{stmt}; \textbf{skip}) = \mathit{stmt}$$

The sequential use of the above laws improves code quality (by making it smaller) and consequently may decrease the program's expected execution time, which is our objective. Our refactorings exploit such a composition of laws, and also exploit AspectJ programming laws [5].

4.1. AspectJ programming laws

In this section we describe our catalogue of aspect-oriented laws. They are written using two side by side boxes, followed by a **provided** clause. This clause introduces conditions, also known as *provisos*, all of which must be true for the the law to be applied. Each proviso is numbered and indicates what must be satisfied when applying the rule from left-to-right. Note that the notation "⇒" indicates that the laws can only be applied from left-to-right. Unlike Cole and Borba's laws, we refer to our laws as unidirectional laws because their intent is for optimization [36], increasing code quality.

4.1.1. Unidirectional laws

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 Cole and Borba's laws ⟨*make aspect privileged*⟩ and ⟨*add empty aspect*⟩ [5]. 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 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. The classical JML compiler (jmlc) [19] always generates 11.0 KB of source code instrumentation, which it compiles to 5.93 KB of bytecode instrumentation, even for empty classes [37].

Law 1. ⟨*remove empty privileged aspect*⟩

```
ts
paspect A {
}
```

⇒

```
ts
```

provided
(1) *A* is not referenced from *ts*. □

Law 2. ⟨*remove before-execution*⟩

```
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
(1) **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**. □

Law 2 shows a transformation which removes the **before** advice when we apply it. 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. The list of fields and methods of class *C* are denoted by *fs* and *ms*, respectively. Moreover, we use *bind*(*context*) to denote the list of advice parameters, including the current executing object (represented by *cthis*). We use the AspectJ designators **this** and **args** to expose such arguments. Thus, in AspectJ, *bind*(*context*) is written as **this(cthis) && args(ps)**. Additionally, we use the AspectJ designator **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*′ contains a reference to the method *m*′, having *cthis* as target. Finally, we use *as* to denote a list of other advice declared in A.

The proviso of **Law** 2 states that when the **before** advice does not add any behavior (or it has an empty body) to the affected method m, we can remove it. Moreover, we can also remove the **before** advice if the declared type is **abstract** or if it is an **interface**. These latter two conditions are valid because the required **within**(C) pointcut designator does not allow the advice to be applied to subtypes. Also since we cannot instantiate a concrete class when we have an **abstract** or an **interface** type, we always can remove such an advice.

This is the simplest law to remove a before advice, it can always be applied when necessary to remove other before advice that can appear in *as*. The derivation of this law is also simple. We apply the Law ⟨*add before execution*⟩ [5, 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. Additionally, our law is unidirectional.

In the context of JML and ajmlc, **Law** 2 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 law, **Law** 3, is similar to the previous one. When it is applied, it removes the **around** advice. The proviso of **Law** 3 is also similar to that of **Law** 2. Thus, when the **around** advice does not insert any behavior, neither before nor after execution of the constrained method m, we can remove such an **around** [2] advice.

According to **Law**'s 3 template, the **around** advice must call the **proceed** method. This call is mandatory because its absence skips the execution of the intercepted join points (e.g., method calls), denoted by jp. For this reason, the developer should be aware that the law's template is only matched (in the left-hand-side) with the call to **proceed** method. We use α preceding a list of context parameters to represent the list of its values. Moreover, jp denotes a list of intercepted join points by the **around** advice in the type C.

As can be observed, **Law** 3 is a variation of **Law** 2. It does not consider intertype declarations and is more general purpose than **Law** 2. Hence, **Law** 3 assumes a set of join points (jp), which belong to type C, that can be captured by the **around** advice. Note that all laws responsible for removing advice can have variations like this one.

We also have similar laws for the remaining three kinds of advice: **after**, **after returning**, and **after throwing** advice (See Table 1). Since those laws are quite similar to the ones described for **before** and **around** advice, we leave their presentation to Appendix B, which is a complement to Sections 4.1 and 5. It shows all laws and refactorings not demonstrated in the referred sections.

[2]**around** advice is a special dynamic crosscutting feature of AspectJ language which encompasses either before and after execution states, both separated by the **proceed** method. Thus, **around** advice can be thought as **before** and **after** advice into a single advice

Law 3. ⟨*remove around-execution*⟩

```
ts                                      ts
class C {                               class C {
  fs                                      fs
  ms                                      ms
}                                       }
paspect A {                      ⇒      paspect A {
  as                                      as
  T around(context) :                   }
      exec(jp) &&
      within(C) &&
      bind(context) {
      body
      proceed(αcontext)
      body′
  }
}
```

provided
(1) **around** advice does not contribute to execution flow of the affected set of join points denoted by jp, or type C is declared **abstract** or it is declared as an **interface**. □

Advice restructuring

There are also laws that help restructuring an advice in order to improve legibility. They are general propose and can be applied to any aspect-oriented system. However, in the context of ajmlc legibility is not an important requirement, because the aspect code generated is not supposed to be handled directly by the developer.

The **Law** 4 is responsible for removing the **this** designator of an advice provided that the parameter t is not used in the advice body. Type T is any type represented by the set of types ts. The notation exp is more general then the execution join point (**exec**) used in previous laws for removing advice. Here, exp might be an execution or call join point, read or write access join points (used for field access), and so forth. In addition, exp represents a compositional set of join points. Hence, join points that are combined by using unary and binary operators can be denoted by exp. We use such generality because the focus of this law is the **this** designator.

Removing the **this** designator from the pointcut expression implies a generalization. This may cause the advice to capture more join points after the transformation. To prevent this, the developer must be aware that the designator, which expose the join point (e.g., **exec** join point), explicitly defines the type of affected join point. For example, to apply this law, consider the affected join point (the execution of $\sigma(C.m)$) of the **Law** 2. As one can observe, its type (C) is explicitly given. If one change the signature of the join point to $\sigma(m)$, this leads to capture every method m's execution hosted by an ordinary class. Thus, the laws' template must be carefully design to perform the correct pattern matching, avoiding undesired behaviors. Finally, note that the **Law** 4 also has similar versions for each kind of advice.

Law 4. ⟨*remove this designator*⟩

```
ts                              ts
paspect A {                     paspect A {
  as                              as
  before(T  t, ps) :        ⇒     before(ps) : exp {
     exp &&                          body'
     this(t) {                     }
     body'                      }
  }
}
```

provided
(1) t is not referenced from *body'* or *exp*. □

Law 5. ⟨*remove within designator*⟩

```
ts                              ts
paspect A {                     paspect A {
  as                              as
  before(context) :               before(context) :
     exp &&                 ⇒        exp &&
     within(T) &&                    bind(context) {
     bind(context) {                 body'
     body'                         }
  }                             }
}
```

provided
(1) The members in the set of join points exposed by *exp* are static or there is no subtype of type T which overrides the affected members. □

For the partial derivation of **Law** 4, we apply the **Law** ⟨*remove this parameter*⟩ [5, Law 14] from left-to-right. Such a law is useful to remove the unused parameter t defined in the template of **Law** 4. Then, we also remove the **this** designator since the type of the intercepted execution is given (as explained before). Hence, the presence of the **this** designator become redundant. In this way, we can safely remove such a designator.

Law 5 is another law used for restructuring an advice. Even though it is similar to **Law** 4, it deals with pointcut designator **within** [3]. When applied, the **Law** 5 is responsible for removing such a designator from the advice's header. To achieve that, the selected points of a program exposed by *exp* must be static. Hence, the advice will not affect new definitions of those members because the owner of static members is the class, not an object instance. Thus, this restriction make the advice not applicable to subtypes even without the **within** designator. On the other hand, if the affected join points in *exp* are non-static members, we must ensure that they have no subtype that overrides them. By respecting these conditions (as stated in the law's proviso), we can argue that the resulting transformation will not change the expected behavior of the program.

Law 5 is fine-grained. It is not a composite law, making the use of derivation unecessary. Additionally, Cole and Borba [5] have no similar law which handles the **within** designator.

Law 6. ⟨*move before advice body to around advice*⟩

```
ts                              ts
paspect A {                     paspect A {
  as                              as
  before(ctx) : exp               before(ctx) : exp
  {                               {
     body                         }
  }                         ⇒     T  around(ctx) : exp{
  T  around(ctx) : exp{              body
     body'                           body'
     proceed(αctx)                   proceed(αctx)
     body''                          body''
  }                               }
}                               }
```

provided
(1) The set of local variables declared by *body*, *body'*, and *body''* are disjoint;
(2) The set of join points, represented by *exp* of the **before** and **around** advice, are the same;
(3) *exp* binds all parameters in *context*; Both **before** and **around** advice use the same parameters within advice body;
(4) *exp* does not execute an **if** pointcut;
(5) The **before** advice has higher precedence than the **around** advice. □

Law 6 is responsible for moving the body from a **before** to an **around** advice. Due to lack of space and for correct indentation, we use *ctx* to denote *context* for short. In relation to law's provisos, the first one ensures that we have no duplicated local variable declarations. So, the set of declared variables of *body*, *body'*, and *body''* must be disjoint in order to avoid conflict after the law's application. The second one states that both advice perform different actions at the same join points, denoted by *exp*. The third proviso states that the advice involved, besides exposing the same join points, must bind the same parameters. Hence, a particular variable used in the body of a **before** advice might also be referred within the **around** advice after transformation.

[3]The AspectJ **within** designator constrains an advice to apply to overridden methods of subtypes.

Table 1: Summary of Aspect-Oriented Laws and Refactorings

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

One important constraint that must be respected is related to join points composition, denoted by *exp*. As aforementioned, we can use unary and binary operators, such as &&, to combine join points in *exp*. Nevertheless, we cannot combine *exp* with an **if** pointcut. Such a pointcut application can potentially disable the execution of the **around** advice, and thus, after law's application, the behavior would be not the same. To prevent this, we provide the fourth proviso for **Law** 6, which does not allow the execution of **if** pointcuts by *exp*, during join points composition. Finally, the fifth proviso states how to apply the transformation in order to preserve the precedence rules imposed on advice before the transformation, and that must be obeyed after the transformation.

Note that the **Law** 6 can have many variations involving different kinds of advice. In the case of the template presented by **Law** 6, we consider a **before** and **around** advice. As illustrated, we move the body of the **before** to the **around** advice. Also, according to the fifth proviso, as the **before** advice has precedence over **around**, the code must be placed just before the call to the **proceed** method and immediately in the beginning of **around** advice. Hence, *body* appears first than *body′*. Assuming now that we have an opposite scenario, the **around** advice having precedence over the **before** advice, the result would be different, having the *body* appearing immediately after the already declared *body′*.

Considering the same scenario of the **Law** 6, but instead of **before** advice, if we have an **after** advice, the result would be moving the *body* to after the call of **proceed** method and immediately before the *body″*. This could be surprising, but according to the rules imposed by aspect precedence, in this case, the **around** advice (more specifically the *body″*) has precedence over the **after** advice declared first (specifically the *body*). Thus, the advice precedence depends on the kind of advice and the ordering which it appears in an aspect. As our laws only treat a single aspect at a time, the precedence is just the advice ordering. But when considering more than one aspect, the precedence is determined explicitly by using the AspectJ `declare precedence` clause.

Still considering **Law** 6, one may observe that the resulting code, after transformation, generates an empty **before** advice. To remove such an empty advice, one must apply the **Law** 2.

Law 7. ⟨*replace method intertype reference with method intertype implementation within advice*⟩

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

provided
(1) m' is not referenced from C, ts, or as. □

Eventually, **Law** 7 replaces the method intertype reference by its implementation. This transformation is useful because just one method intertype is only referenced by one advice (respecting the law's proviso). As discussed in the previous law, the resulting code seems to be useless, since they keep the intertype method which is no longer used. This happens because the intent of the law is just to replace the intertype reference

```
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.

with its implementation. To remove such an unused intertype method we apply the **Law** 11, which removes it. As outlined in the next section, we discuss that the application of both **Laws** 7 and 11 are useful to derive the Refactoring 1, which inlines a method intertype within advice.

Soundness of laws

As aforementioned, programming laws [11] 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's work [20]. They present a proof sketch for one of their laws (*add before-execution*) based on a formal semantics of an AOP language [38], which is similar to AspectJ. Furthermore, they argue that their solution allows the proof for other five laws [20]. As some of our laws are justified by means of the application of their laws, we rely on their correctness proofs for their laws.

However, there are laws [4] 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 [38] to prove that these laws are behavior-preserving transformations.

Even though we have some laws [5] 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.

5. Deriving AspectJ refactorings

In this section we describe our catalogue of aspect-oriented refactorings. Likewise our laws, they are written using two side by side boxes, followed by provisos (**provided** clause) in order to apply them. Furthermore, as with laws, the notation "⇒" indicates that the refactorings we present are unidirectional (they are only applied from left-to-right).

[4]Except by **Law** 1, which we derived completely from [5, 20], the other laws and refactorings are proposed by this work.

[5]**Law** 3, 7, 8, 10, and 11.

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

```
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
(1) m' is not referenced from C, ts, or as. □

5.1. Unidirectional refactorings

The first AO refactoring we present, **Refactoring** 1, is a refactoring that inlines the method intertype implementation within the **before** advice. This transformation is useful because the method intertype is only referenced by just one advice. The transformation removes the method intertype and moves its implementation to the advice (**before** advice). The derivation of this refactoring involves two simple laws. Consider them step by step: (1) apply **Law** 7 ⟨*replace method intertype reference with method intertype implementation within advice*⟩, replacing all references of the method intertype m' within **before** advice with its implementation, and (2) apply **Law** 11 ⟨*remove method intertype implementation*⟩, removing the method intertype m'.

Refactoring 1 is useful for ajmlc optimization when, for example, a **before** advice is checking a precondition and makes a reference to 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 safe to remove it through 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.

The template of **Refactoring** 1 shows the transformation concerning a **before** advice, but there are variations of such a refactoring that can deal with other kinds of AspectJ advice [39].

Refactoring 2. ⟨*merge distinct advice*⟩

```
ts
paspect A {
 as
  T around(ctx) : exp{
     body′
     proceed(αctx)
     body′′
  }
  before(ctx) : exp {
     body
  }
}
```

⇒

```
ts
paspect A {
 as
  T around(ctx) : exp{
     body′
     body
     proceed(αctx)
     body′′
  }
}
```

provided
(1) The set of local variables declared by *body*, *body*′, and *body*′′ are disjoint;
(2) The set of join points, represented by *exp* of the **around** and **before** advice, are the same;
(3) *exp* binds all parameters in *ctx*; Both **around** and **before** advice use the same parameters within advice body;
(4) *exp* does not execute an **if** pointcut;
(5) The **around** advice has higher precedence than the **before** advice. □

Refactoring 2 merges a **before** advice with an **around** advice. This is possible since both advice intercept the same set of join points, denoted by *exp* (second proviso). The first proviso ensures that we have no duplicated local variable declaration. In other words, the set of declared local variables from **before** and **around** advice are disjoint. This avoids conflicts when moving *body* from **before** to **around** advice's body. The third proviso states that both advices must bind the same parameters referred in advices' body. Hence, *exp* is responsible for binding all parameters in *ctx*. In addition, the fourth proviso, as discussed in Section 4.1, constrains the execution of **if** pointcuts.

The fifth proviso states that the **around** advice has higher precedence over the **before** advice. This is due to advice precedence imposed by AOP languages such as AspectJ. As a result, we can observe that the **before** advice's body (*body*) was placed immediately after the existing code denoted by *body*′ (as illustrated by the right-hand-side of the refactoring template).

For the derivation of **Refactoring** 2, we apply **Laws** 6 and 2, respectively. **Law** 6 is used just to move the **before** advice body (represented by *body*) to **around** advice. Then, the useless **before** advice, which now has an empty body, is removed by applying the **Law** 2. As previously discussed, a programming law may involve different advices. Moreover, the advice precedence depends on the kind of advice and the ordering which it appears in an aspect. This fact is exemplified in the **Refactoring** 2 template, which changes the advice order observed in the template of **Law** 6. We performed this change only to emphasize the variations that a law or refactoring might present.

Refactoring 3. ⟨*split around into before, after returning, and after throwing*⟩

```
ts
paspect A {
 as
  T around(ctx) : exp{
     try {
        body
        proceed(αctx)
        body′
     } catch(E e) {
        body′′
     }
  }
}
```

⇒

```
ts
paspect A {
 as
  before(ctx) : exp{
     body
  }
  after(ctx)
  returning(T t) : exp{
     body′
  }
  after(ctx)
  throwing(E e) : exp{
     body′′
  }
}
```

provided
(1) *body*′ and *body*′′ do not use local variables declared by *body*;
(2) *body* is localized before the call to **proceed** method;
(3) *body* does not change the values of context variables in *ctx*. □

In the context of aspects generated by ajmlc compiler, the **Refactoring** 2 is useful, for example, to merge a precondition and a postcondition into a single aspect construct. In this way, when we have a static method annotated with JML, which generates a **before** advice for precondition checking and one **around** advice for postcondition checking, we can merge them into a single advice, resulting in a single **around** advice for both precondition and postcondition checking.

It is worth noting that the **Refactoring** 2 is only applied to static methods since the **around** advice checks JML postconditions in a conjunction way (when we have inherited specifications with overridden methods). And, since JML checks all preconditions in a disjunction form, the application of this refactoring for non-static methods might break JML semantics for precondition checking. So, such a refactoring is still valid for join points that are non-static in an aspect-oriented systems which are non-ajmlc based.

Refactoring 4. ⟨*split around into after returning and after throwing*⟩

```
ts                                ts
paspect A {                       paspect A {
  as                                as
  T around(ctx) : exp{              after(ctx)
      try {                         returning(T  t) : exp{
          proceed(αctx)                 body
          body                  ⇒   }
      } catch(E  e) {               after(ctx)
          body′                     throwing(E  e) : exp{
      }                                 body′
    }                               }
}                                 }
```

provided
(→) *body* and *body*′ do not use local variables declared before the **proceed** method. □

The **Refactoring** 3 splits the **around** advice into three distinct advice: (i) **before**, (ii) **after returning**, and (iii) **after throwing**. We can apply such refactoring only if *body*′ and *body*′′ do not refer to local variables declared by *body* (first proviso). The piece of code *body* is localized immediately before the call to **proceed** method (second proviso). Without this dependency, one is allowed to replace the **around** advice, in the left-hand-side of the template, for one **before** advice and two kinds of **after** advice: **after returning** and **after throwing**. Then, each part of the **around** advice's body is placed in its respective advice.

The third proviso states that the *body* cannot modify the values of context variables denoted by *ctx*. This constraint is valid since once *body* changes the values of context variables, both *body*′ and *body*′′ refer to modified values. This collateral effect cannot be reproduced when we separate each body with its respective advice. So, to safely apply the transformations described in the **Refactoring** 3, such a collateral effect must be avoided.

To derive **Refactoring** 3, we have to define new programming laws to add advices. To do so, we adapted laws from Cole and Borbas work [5], which were originally developed to restructure OO code. We modified them to restructure AO code. The new programming laws is then applied in the following order to derive the **Refactoring** 3: (1) apply the **Law** ⟨*add before-execution*⟩ [5, Law 3] from left-to-right; this law introduces a **before** advice which modularizes the code represented by *body*, (2) apply **Law** ⟨*add after-execution returning successfully*⟩ [5, Law 7], from left-to-right, to modularize *body*′ into an **after returning** advice; *body*′ denotes the code used for normal termination of the affected join points *exp*, and (3) apply **Law** ⟨*add after-execution throwing exceptions*⟩ [5, Law 10], from left-to-right, to modularize *body*′′ within **after throwing** advice; *body*′′ denotes the code used for exceptional termination of the affected join points *exp*.

In relation to ajmlc, **Refactoring** 3 is useful when we have, for example, a static method with preconditions and the two kinds of JML postconditions [25]: normal postcondition and exceptional postcondition. In this scenario, the aspects generated to check those JML features is somewhat equivalent to **Refactoring**'s 3 left-hand-side. If we have postcondtions that do not consider JML *old expressions* [25], we can perform the transformation mechanics of **Refactoring** 3.

JML *old expressions* refer to pre-state variables (class fields) in postconditions. Thus, we need to save state variables before method's execution (pre-state) and after method's execution (post-state). Afterwards, we can refer to pre-state variables in postconditions. The **around** advice is used to handle situations like that. The **proceed** method is responsible for separating each state. Before its call, we have the pre-state, after its call, we have the post-state. Such a dependency, imposed by JML old expressions, breaks the first proviso of **Refactoring** 3. So, if we can guarantee that postconditions are free from old expressions, we do not have a coupling between the states during a method execution. In this way, we can safely apply such a refactoring, which decouples them into different advices.

Refactoring 4 is slightly different from previous one. When applied, from left-to-right, it splits the **around** advice only into **after returning** and **after throwing**. The single proviso states that both *body* and *body*′ cannot use local variables declared before the call to **proceed** method. As discussed, this implies in a dependency, and hence, we cannot split the **around** advice. The derivation of **Refactoring** 4 is similar to previous one. Since we do not address, in this paper, laws for adding advice, we use an adaptation of programming laws from Cole and Borba's work [5]. We slightly modified the laws to give support to refatoring of AO code. Then, we can apply **Law** ⟨*add after-execution returning successfully*⟩ [5, Law 7] and **Law** ⟨*add after-execution throwing exceptions*⟩ [5, Law 10], respectively. Both laws are applied from left-to-right, and modularize *body* within an **after returning** and *body*′ within an **after throwing** advice.

Regarding ajmlc, the template of **Refactoring** 4 applies when we have a method in which only normal and exceptional postconditions are specified. As aforementioned, if those postconditions do not use JML old expressions, the **around** advice generated by ajmlc can be divided into the two kinds of **after** advice. This is possible since the absence of JML old expressions does not cause a dependency between pre- and post-states (proviso), which can be only handled with **around** advice.

Other refactorings

Besides the refactorings presented above, we have identified others that can be useful for restructuring AO code. For instance, **Refactoring** 5 is a variation of the **Refactoring** 3. For this reason, we leave its presentation to Appendix B. The main difference is that the template of **Refactoring** 5 assumes the same code (*body*′) to be executed either in normal and abnormal termination (by throwing exceptions). In this way, the resulting code uses only an **after** advice to encapsulate *body*$'$. We use a single **after** advice because when a method terminates normally or abnormally, it is always executed. The provisos required to apply **Refactoring** 5 are the same provided by **Refactoring** 3. Moreover, to derive **Refactoring** 5, we use ⟨*add after-execution*⟩ [5, Law 3] and ⟨*add after-execution*⟩ [5, Law 5], respectively. Note that we assume that those laws are adapted to handle aspect-oriented constructs.

In relation to **Refactoring** 6, we leave its presentation to Appendix B because we cannot derive it using our programming laws or employing laws from Cole and Borba. In Section 5, we present all refactorings that can be derived by using aspect-oriented programming laws described in this paper or by applying programming laws proposed in related works. Hence, we need specific new laws to derive **Refactoring** 6. **Refactoring** 6 was created to extract duplicated code that cuts across several members (two distinct advice in this case) of an aspect. It then modularizes such a duplicated code in a method inside an aspect. Note that this aspect method is not an intertype declaration, and thus cannot be used outside the aspect in which it is declared.

The code generated by the ajmlc compiler has no **around** advice that matches with the template defined in the **Refactoring** 5. This is a consequence of the semantics of the JML language. However, JML semantics is being modified in the OpenJML project [6]. The semantic of OpenJML will impose modifications in the ajmlc compiler. The aspects generated by the new compiler will include different patterns of **around** advices. It opens opportunities to apply the **Refactoring** 5. On the other hand, there are many cases where the **Refactoring** 6 can be applied to aspects generated through the current version of ajmlc compiler.

6. Discussion

We showed that aspect-oriented code can be more cleaner and more legible, increasing the overall systems maintainability. Even though we use the aforementioned catalog of laws and refactorings in terms of compiler optimization (ajmlc [16]), we claim that the benefits are general enough to be applied in any aspect-oriented system. In addition, the transformations we provided, in almost all cases, reduced the number of lines of code and consequently improve the final bytecode size and running time of the system being restructured. Regarding these two quantitative methods, bytecode size and running time is discussed and demonstrated by means of an empirical study in Section 7.

We have argued that the transformations we conduct in Sections 4.1 and 5 results in a reduced code with less number of lines and consequently a smaller bytecode size. Hence, ideally, having one piece of advice rather than two reduces the complexity of the aspect code. However, this reasoning is not too straightforward. Depending on the scenario, we can have aspects with more lines of code that generate smaller bytecode than ones with less lines of code that generate bigger bytecode. The answer to this instability is in the aspect-weaving mechanism. As an example, consider a scenario when we have two aspects A and A'. The former declares a single **around** advice and the latter declares two pieces of advice, **before** and **after** advice. After weaving, we can observe that, surprisingly, the second aspect (A'), even declaring two advice, has a smaller bytecode in relation to the first one A. This result depends on what weaver is used for aspect composition.

The main issue behind **around** advice weaving is the inefficiency in its instrumentation, resulting in a bigger bytecode than the one which does not use it. Hence, we have a trade-off. On the one hand, when we employ **around** advice, we have fewer lines of code, but these result in a larger bytecode size after weaving. On the other hand, we have more lines of code and smaller bytecode size, when employing **before** and **after** advice. Nevertheless, we can obtain different results if we apply a weaver in which has optimization strategies for **around** advice that reduce the generated bytecode size. Therefore, those aforementioned laws and refactorings can also be used to improve the AO system. Further information and discussions refer to Section 7. There, we consider two existing AspectJ weavers (ajc and abc [40]).

Another important issue to consider is related to system evolvability. The transformations based on our AO laws and refactorings are not guaranteed to be correct when new features are added to a system (e.g., new types with new methods and overridden methods). Thus the transformed code could have inconsistencies when adding such new features to a system. That is, our laws are, in some cases, only designed for closed programs—as we did not consider evolvability. This means that when changing the set of types or methods in a program, the ajmlc tool will have to be run again to regenerate all the runtime assertion checking code, which should not be too much of a burden. However, this limitation makes it dangerous to use our laws and transformations by hand on an evolving system. Thus, depending on the scenario, the developer may even need to undo some transformations. For instance, suppose the developer applies **Law** 5, to remove a **within** designator, by respecting the second part of its proviso. If the developer later adds a subtype of type T, then earlier application of the law is not valid with respect to this new system, since the advice, in which its **within** designator was removed, would have affected methods in T that T overrides from its supertype. Future work would be needed to make these laws valid for evolvability, or to describe the kinds of system evolution that are valid for earlier applications of certain laws.

7. Case Study

This section conducts a case study to evaluate the benefits and limitations of the optimization mechanisms proposed in this article.

7.1. Study Settings

This subsection describes assessment procedures of our empirical study.

[6]http://sourceforge.net/apps/trac/jmlspecs/wiki/OpenJml

Table 2: Quantification of Laws and Refactorings in the JAccounting, JSpider, Prevayler, and Bomber systems

	JAccounting	JSpider	Prevayler	Bomber
	Qty	Qty	Qty	Qty
Law 1	28	148	69	5
Law 2	33	27	46	10
Law 3	11	249	126	20
Law 4	34	7	1	2
Law 5	34	7	1	2
Law 6	8	9	8	3
Law 7	2	95	40	2
Law 9	30	36	57	-
Law 11	5	115	65	2
Refactoring 1	2	94	39	2
Refactoring 2	8	9	2	3
Refactoring 3	3	43	21	-
Refactoring 4	11	249	126	20
Total (T)	**209**	**1,088**	**601**	**71**

The study explores four different applications annotated with JML specifications [25]. Three of them are pure Java programs and one is a Java ME application. The case study takes into account a Java ME application, called Bomber [7] We do so because, unlike jmlc, our ajmlc compiler generates code compliant with the Java ME platform [16]. Bomber is a simple software product line game based on Java ME MIDP 2.0. The case study also involves the JAccounting [8], JSpider [9], Prevayler [10]. We believe that these applications are representative of how Design by Contract is typically used to ensure functional software correctness in real software development efforts.

After annotating the four programs with JML, we compiled them using both the classic JML compiler jmlc [19] and the ajmlc [16]. The latter compiler implements the optimizations proposed in this paper. We employed the latest versions of each compiler: jmlc-5.5 [11] and ajmlc-1.1 [12]. For ajmlc we evaluated two versions: with and without the laws and the refactorings (optimizations) proposed in this work. In addition, we used ajmlc with two different weaving processes: the standard AspectJ compiler (ajc), and abc [40]. We considered the ajc-1.3 [13] and the latest version of abc (abc-1.3 [14]). The difference is that the abc weaver itself includes various optimizations.

In the measurement process, the data was partially gathered by the Tracing and Profiling Eclipse Plugin [15]. It addresses the tracing and profiling phases of the application lifecycle.

We restricted the performance analysis to the methods annotated with JML specifications. Thus, from the set of methods annotated with JML, we randomly chose three methods of each application. Using the profile information, we collected the mean execution time of one thousand executions of each method. The values are given in milliseconds.

The quantitative assessment was based on three metrics: instrumented source code size (ISC), instrumented bytecode size (Bytecode), and execution time. ISC is calculated for the code generated through both jmlc and ajmlc compilers. It encompasses only the intermediate source code produced. ISC is useful to assess the amount of additional code necessary to translate the JML specifications into automatic runtime checks. Bytecode, in turn, encompasses the final Java bytecode with such extra code for checking contracts during runtime.

7.2. Study Results

This subsection presents the results of the measurement process. The data have been collected based on the assessment procedures and metrics described in subsection 7.

Table 2 summarizes the number of laws and refactorings applied by ajmlc compiler to optimize each target system. As can be observed, for the Jspider application, ajmlc's optimizer applies a total of 1,088 transformations to the aspects generated by original ajmlc compiler. On the other hand, only 71 transformations are used to optimize the Bomber application. **Law** 3 is applied with higher frequency in three applications:JSpider, Prevayler, and Bomber. Recall that **Law** 3 is responsible for removing the **around** advice,

Table 3 presents the results for code size performance metrics: instrumented source code size (ISC) and instrumented bytecode size (Bytecode). For the ajmlc compiler, the Bytecode metric is calculated using two different weavers: *ajc* and *abc*. For both metrics, code size is measured in megabytes (MB). We applied the paired t-test to compare the values in Table 3 and evaluate if there are statistically significant difference between the compared results. We drew the following conclusions: 1) the source code produced by ajmlc compiler is smaller than the code generated by the jmlc compiler; 2) the optimizations reduce the ISC; 3) the optimizations reduces the bytecode size

[7]http://j2mebomber.sourceforge.net.
[8]https://jaccounting.dev.java.net.
[9]http://j-spider.sourceforge.net/.
[10]http://www.prevayler.org/.
[11]http://sourceforge.net/projects/jmlspecs/
[12]http://www.cin.ufpe.br/ hemr/JMLAOP/ajmlc.htm
[13]http://www.eclipse.org/aspectj/downloads.php
[14]http://abc.comlab.ox.ac.uk/dists/1.3.0/package
[15]2http://www.eclipse.org/projects/project summary.php?projectid=tptp.performance

Table 3: Code size measurements

Application	Original (MB)	Optimized (MB)	Decrease (%)	Application	Original (MB)	Optimized (MB)	Decrease (%)
JAccounting				**JSpider**			
– ISC –				*– ISC –*			
jmlc	5.42	-	-	jmlc	10.0	-	-
ajmlc	1.28	1.25	2.34	ajmlc	1.93	1.85	4.15
– Bytecode –				*– Bytecode –*			
jmlc	2.14	-	-	jmlc	4.10	-	-
ajmlc(ajc)	4.71	3.11	33.97	ajmlc(ajc)	7.81	5.59	28.43
ajmlc(abc)	1.56	1.29	17.31	ajmlc(abc)	2.56	2.17	15.23
Prevayler				**Bomber**			
– ISC –				*– ISC –*			
jmlc	3.29	-	-	jmlc	-	-	-
ajmlc	0.90	0.87	3.33	ajmlc	0.89	0.79	11.24
– Bytecode –				*– Bytecode –*			
jmlc	1.29	-	-	jmlc	-	-	-
ajmlc(ajc)	3.65	2.54	30.41	ajmlc(ajc)	2.93	2.01	31.40
ajmlc(abc)	1.25	1.05	16.00	ajmlc(abc)	1.07	0.86	19.63

***Denotes an updated measurement [1]**

Table 4: Running time measurements

Method	**Original** (msec) jmlc	ajmlc ajc	ajmlc abc	**Optimized** (msec) ajmlc ajc	ajmlc abc	**Decrease** (%) 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	2.14*	1.94*	0.26*	0.20*	0.17*	89.60*	34.61*
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

produced by the ajmlc compiler - this result was confirmed for both *ajc* and *abc* weavers; 4) when the *ajc* weaver is used, the byte code produced by the jmlc compiler is smaller than that generated by the ajmlc compiler - this result is observed even when the optimizations are applied; 6) when the *abc* weaver is employed, the ajmlc compiler produces a far smaller byte code when compared with the jmlc compiler.

Concerning the execution time (see Table 4), we observed that the optimized ajmlc produced code that executes faster than the non-optimized version. Moreover, the running time is greatly reduced when the optimized ajmlc employs the abc weaver. Additionally, the execution time of the ajmlc aspects code is faster than the jmlc code, even without our optimizations. Such bad performance in jmlc is due to many reflective calls in the jmlc generated code. These conclusions were statistically attested through the application of the paired t-test.

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 [16]. Thus, we cannot execute jmlc generated code for the Bomber program with the Java ME API that it uses.

8. Conclusions

In this paper, we have presented a catalog of programming laws and refactorings 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 aspect-oriented 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.

Appendix 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/scp10.

Appendix B. Laws and Refactorings

Law 8. ⟨*remove after-execution*⟩

```
ts
paspect A {
 as
 after(context) :
   exec(jp) &&
   within(T) &&
   bind(context) {
   body
 }
}
```

⇒

```
ts
paspect A {
 as
}
```

provided
(1) **after** advice does not contribute to execution flow of the set of affected join points *jp*, or type *T* is declared **abstract** or it is declared as an **interface**. □

Law 9. ⟨*remove after-execution returning*⟩

```
ts
paspect A {
 as
 after(context)
   returning(T t) :
   exec(jp) &&
   within(T') &&
   bind(context) {
   body
 }
}
```

⇒

```
ts
paspect A {
 as
}
```

provided
(1) **after returning** advice does not contribute to execution flow of the set of affected join points *jp*, or type *T*′ is declared **abstract** or it is declared as an **interface**. □

Law 10. ⟨*remove after-execution throwing*⟩

```
ts
paspect A {
 as
 after(context)
   throwing(E e) :
   exec(jp) &&
   within(T) &&
   bind(context) {
   body
 }
}
```

⇒

```
ts
paspect A {
 as
}
```

provided
(1) **after throwing** advice does not contribute to execution flow of the set of affected join points *jp*, or type *T* is declared **abstract** or it is declared as an **interface**. □

Law 11. ⟨*remove method intertype implementation*⟩

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

⇒

```
ts
paspect A {
 as
}
```

provided
(1) *m*′ is not referenced from *ts*, or *as*. □

Refactoring 5. ⟨*split around into before and after*⟩

```
ts
paspect A {
 as
 T around(ctx) : exp{
    try {
      body
      proceed(αctx)
      body'
    } catch(E e) {
      body'
    }
  }
}
```

⇒

```
ts
paspect A {
 as
 before(ctx) : exp {
   body
 }
 after(ctx) : exp {
   body'
 }
}
```

provided
(1) *body*′ does not use local variables declared by *body*; *body* is localized before the call to **proceed** method;
(2) *body* does not change the values of context variables in *ctx*. □

Refactoring 6. ⟨*extract aspect method*⟩

```
ts
paspect A {
  as
  before(ctx) : exp {
     body
     body'
  }
  after(ctx) : exp {
     body''
     body
  }
}
```

⇒

```
ts
paspect A {
  as
  before(ctx) : exp {
     m(αps)
     body'
  }
  after(ctx) : exp {
     body''
     m(αps)
  }
  T m(ps) {
     body
  }
}
```

provided
(1) The set of local variables declared by *body*, *body*$'$, and *body*$''$ are disjoint;
(2) *body* does not use context variables in *ctx*. □

Acknowledgements

This work was partially supported by FACEPE, FINEP, and Cenpes/Petrobras. 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

[1] H. Rebêlo, R. Lima, M. Cornélio, G. T. Leavens, A. Mota, C. Oliveira, Optimizing JML features compilation in ajmlc using aspect-oriented refactorings, in: SBLP '09: Proceedings of the 2009 Brazilian Symposium on Programming Languages, Brazilian Computer Society, 2009, pp. 117–130.

[2] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, Jean-MarcLoingtier, J. Irwin, Aspect-oriented programming, in: European Conference on Object-Oriented Programming (ECOOP) , Jyvskyl, Finland, no. 1241 in Lecture Notes in Computer Science, Springer-Verlag, 1997, pp. 220–242.

[3] D. Binkley, M. Ceccato, M. Harman, F. Ricca, P. Tonella, Automated refactoring of object oriented code into aspects, in: ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance, IEEE Computer Society, Washington, DC, USA, 2005, pp. 27–36. doi:http://dx.doi.org/10.1109/ICSM.2005.27.

[4] J. Hannemann, G. C. Murphy, G. Kiczales, Role-based refactoring of crosscutting concerns, in: AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development, ACM, New York, NY, USA, 2005, pp. 135–146. doi:http://doi.acm.org/10.1145/1052898.1052910.

[5] L. Cole, P. Borba, Deriving refactorings for aspectj, in: AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development, ACM, New York, NY, USA, 2005, pp. 123–134. doi:http://doi.acm.org/10.1145/1052898.1052909.

[6] W. F. Opdyke, Refactoring object-oriented frameworks, Ph.D. thesis, Champaign, IL, USA (1992).

[7] D. B. Roberts, Practical analysis for refactoring, Ph.D. thesis, Champaign, IL, USA, adviser-Johnson, Ralph (1999).

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

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

[10] M. P. Monteiro, J. M. Fernandes, Towards a catalogue of refactorings and code smells for aspectj (2006) 214–258doi:http://dx.doi.org/10.1007/11687061_7.

[11] C. A. R. Hoare, I. J. Hayes, H. Jifeng, C. C. Morgan, A. W. Roscoe, J. W. Sanders, I. H. Sorensen, J. M. Spivey, B. A. Sufrin, Laws of programming, Commun. ACM 30 (8) (1987) 672–686. doi:http://doi.acm.org/10.1145/27651.27653.

[12] A. W. Roscoe, C. A. R. Hoare, The laws of occam programming, Theor. Comput. Sci. 60 (2) (1988) 177–229. doi:http://dx.doi.org/10.1016/0304-3975(88)90049-7.

[13] R. Bird, O. de Moor, Algebra of programming, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997.

[14] S. Seres, M. Spivey, T. Hoare, Algebra of logic programming, in: Proceedings of the 1999 international conference on Logic programming, Massachusetts Institute of Technology, Cambridge, MA, USA, 1999, pp. 184–199.

[15] P. Borba, A. Sampaio, A. Cavalcanti, M. Cornélio, Algebraic reasoning for object-oriented programming, Sci. Comput. Program. 52 (1-3) (2004) 53–100. doi:http://dx.doi.org/10.1016/j.scico.2004.03.003.

[16] H. Rebêlo, S. Soares, R. Lima, L. Ferreira, M. Cornélio, Implementing java modeling language contracts with aspectj, in: SAC '08: Proceedings of the 2008 ACM symposium on Applied computing, ACM, New York, NY, USA, 2008, pp. 228–233. doi:http://doi.acm.org/10.1145/1363686.1363745.

[17] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, E. Poll, An overview of JML tools and applications, International Journal on Software Tools for Technology Transfer (STTT) 7 (3) (2005) 212–232.
URL http://dx.doi.org/10.1007/s10009-004-0167-4

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

[19] Y. Cheon, G. T. Leavens, A runtime assertion checker for the Java Modeling Language (JML), in: H. R. Arabnia, Y. Mun (Eds.), Proceedings of the International Conference on Software Engineering Research and Practice (SERP '02), Las Vegas, Nevada, USA, June 24-27, 2002, CSREA Press, 2002, pp. 322–328.

[20] L. Cole, P. Borba, A. Mota, Proving aspect-oriented programming laws, in: G. T. Leavens, C. Clifton, R. Lämmel (Eds.), Foundations of Aspect-Oriented Languages, 2005.

[21] B. Meyer, Applying "design by contract", Computer 25 (10) (1992) 40–51. doi:http://dx.doi.org/10.1109/2.161279.

[22] L. C. Briand, W. J. Dzidek, Y. Labiche, 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), IEEE Computer Society, Washington, DC, USA, 2005, pp. 687–690. doi:http://dx.doi.org/10.1109/ICSM.2005.55.

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

[24] D. Wampler, Contract4J for Design by Contract in Java: Design Pattern-Like Protocols and Aspect Interfaces, in: ACP4IS Workshop at AOSD 2006, 2006, pp. 27–30.

[25] G. T. Leavens, A. L. Baker, C. Ruby, Preliminary design of JML: A behavioral interface specification language for Java, ACM SIGSOFT Software Engineering Notes 31 (3) (2006) 1–38.

[26] S. Hanenberg, C. Oberschulte, R. Unl, Refactoring of aspect-oriented software, in: NODe '03: Proceedings of the 4th Annual International Conference on Object-Oriented and Internet-based Technologies, Concepts, and Applications for a Networked World (Net.ObjectDays), 2003, pp. 19–35.

[27] J. Wloka, Refactoring in the presence of aspects, in: 13th Workshop for PhD Students in Object-Oriented Systems (PhDOOS), at ECOOP '03, 2003.

[28] M. P. Monteiro, a. M. Fernandes, Jo Towards a catalog of aspect-oriented refactorings, in: AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development, ACM, New York, NY, USA, 2005, pp. 111–122. doi:http://doi.acm.org/10.1145/1052898.1052908.

[29] M. Marin, L. Moonen, A. van Deursen, An approach to aspect refactoring based on crosscutting concern types, SIGSOFT Softw. Eng. Notes 30 (4) (2005) 1–5. doi:http://doi.acm.org/10.1145/1082983.1083140.

[30] F. C. Filho, N. Cacho, E. Figueiredo, R. Maranh ao, A. Garcia, C. M. F. Rubira, Exceptions and aspects: the devil is in the details, in: SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering, ACM, New York, NY, USA, 2006, pp. 152–162. doi:http://doi.acm.org/10.1145/1181775.1181794.

[31] F. C. Filho, A. Garcia, C. M. F. Rubira, Extracting error handling to aspects: A cookbook, in: ICSM, 2007, pp. 134–143.

[32] S. Soares, P. Borba, E. Laureano, Distribution and persistence as aspects, Softw. Pract. Exper. 36 (7) (2006) 711–759. doi:http://dx.doi.org/10.1002/spe.v36:7.

[33] S. Soares, P. Borba, Towards reusable and modular aspect-oriented concurrency control, in: SAC '07: Proceedings of the 2007 ACM symposium on Applied computing, ACM, New York, NY, USA, 2007, pp. 1293–1294. doi:http://doi.acm.org/10.1145/1244002.1244281.

[34] J. Hannemann, G. Kiczales, Design pattern implementation in java and aspectj, SIGPLAN Not. 37 (11) (2002) 161–173. doi:http://doi.acm.org/10.1145/583854.582436.

[35] A. Garcia, C. Sant'Anna, E. Figueiredo, U. Kulesza, C. Lucena, A. von Staa, Modularizing design patterns with aspects: a quantitative study, in: AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development, ACM, New York, NY, USA, 2005, pp. 3–14. doi:http://doi.acm.org/10.1145/1052898.1052899.

[36] A. Sampaio, An Algebraic Approach to Compiler Design, World Scientific, 1997.

[37] H. Rebêlo, S. Soares, R. Lima, P. Borba, M. Cornélio, 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), no. CS-TR-08-07 in Technical Report, School of EECS, UCF, 4000 Central Florida Blvd., Orlando, Florida, 32816-2362, 2008, pp. 11–18.

[38] R. Lämmel, A semantical approach to method-call interception, in: Proceedings of the 1st international conference on Aspect-oriented software development, ACM, New York, NY, USA, 2002, pp. 41–55. doi:10.1145/508386.508392.

[39] R. Laddad, AspectJ in Action: Practical Aspect-Oriented Programming, Manning Publications Co., Greenwich, CT, USA, 2003.

[40] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, J. Tibble, abc: an extensible aspectj compiler, in: AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development, ACM, New York, NY, USA, 2005, pp. 87–98. doi:http://doi.acm.org/10.1145/1052898.1052906.