Design by Contract for Aspects, by Aspects

Tim Molderez [*] Dirk Janssens
Dept. of Mathematics and Computer Science
University of Antwerp, Belgium
{tim.molderez,dirk.janssens}@ua.ac.be

Abstract

Run-time contract enforcement is a useful means to help ensure the reliability of a software system. Due to the scattering and tangling nature of crosscutting concerns, aspects can have a high degree of coupling with other modules. Contract enforcement should therefore prove especially useful for aspects. This paper presents such a run-time enforcement algorithm for a minimal aspect-oriented language, guided by the advice substitution principle: an aspect-oriented version of Liskov substitution. As contract enforcement in itself is a crosscutting concern, the algorithm is specified using aspects as well.

Categories and Subject Descriptors F.3.1 [*Theory of Computation*]: Specifying and Verifying and Reasoning about Programs

General Terms Design, Languages, Reliability

Keywords run-time contract enforcement, substitutability, aspect-oriented programming

1. Introduction

Aspect-oriented programming (AOP) languages have introduced powerful mechanisms to be able to modularize crosscutting concerns, as it allows for the modification of a program's behaviour in a quantifiable manner. However, while quantification allows for crosscutting concerns to be expressed as separate modules, i.e. aspects, the difficulty of verifying whether an aspect always performs correctly is quantified as well. This is evidenced by the large body of work that currently exists in the field of aspect-oriented program verification. This paper focuses on AOP from a design by contract (DbC) perspective. The essential idea is that, whenever a method call is made, its preconditions and corresponding class invariants must be met in order for its postconditions to hold. We are interested in how this principle translates itself to AOP.

In object-oriented programming, it is the responsibility of the caller to ensure that the preconditions and invariants are met. However, aspects follow the inversion of control principle. That is, an aspect decides for itself when an advice is "called". When aspects have contracts, this means that the aspect itself should ensure that the preconditions and invariants are met whenever an advice is executed. Additionally, its postconditions should not break the postconditions of the method call being advised.

This responsibility is quite similar for subclass methods overriding another method. Because the method's caller may be oblivious towards the dynamic type of a method call's receiver, the overriding method is responsible for complying with the expectations of the caller. The Liskov substitution principle [3] ensures this compliance by requiring that preconditions may not be strengthened in a subclass, postconditions may not be weakened and invariants must be preserved. This paper discusses how Liskov substitution can be easily adapted for aspects, resulting in the advice substitution principle. A contract enforcement algorithm is then presented, which applies the advice substitution principle to a minimal aspect-oriented language called ContractAJ. Both this language and enforcement algorithm are based on Findler et al. [2], where contract enforcement is implemented for ContractJava, a minimal object-oriented language. Whereas ContractJava makes use of wrapper methods to implement contract enforcement, the implementation in ContractAJ purely uses aspects and therefore does not need to make any modifications to the program, as contract enforcement is a prime example of a crosscutting concern.

The remainder of this paper is structured as follows: Sec. 2 presents the syntax and semantics of ContractAJ. Sec. 3 then discusses the advice substitution principle. The contract enforcement algorithm is introduced in Sec. 4. Sec. 5 presents related work; Sec. 6 concludes the paper and discusses future work.

2. ContractAJ

The aspect-oriented language in which our contract enforcement algorithm is expressed is called ContractAJ, based on the object-oriented ContractJava language introduced in Findler et al. [2].

2.1 Syntax

The syntax of ContractAJ is shown in Fig. 1. What is noticeable immediately is that we have not introduced a separate module type dedicated to aspects, i.e. there is no `aspect` keyword. Similar to Classpects [5], definitions of pointcuts and advice are allowed in regular classes, such that they can effectively serve as aspects as well. We made this choice as it makes the contract enforcement algorithm more concise, as aspects no longer need to be treated as a different case.

[*] Funded by a doctoral scholarship of the Research Foundation - Flanders (FWO)

$$
\begin{array}{rcl}
\mathit{program} & ::= & \mathit{def}^{*}\,\mathit{expr} \\
\mathit{def} & ::= & \textbf{class}\ c\ \textbf{extends}\ \{\mathit{field}^{*}\ \mathit{meth}^{*}\ \mathit{adv}^{*}\} \\
\mathit{field} & ::= & t\ [c.]f \\
\mathit{method} & ::= & t\ [c.]m\ (\mathit{arg}*)\{\mathit{expr}\}\mathit{contract} \\
\mathit{adv} & ::= & t\ \mathit{advt}\ a\ (\mathit{arg}^{*})\ :\ \mathit{prec}?\ \mathit{pcut}\{\mathit{expr}\}\mathit{contract} \\
\mathit{contract} & ::= & @\textbf{pre}\{\mathit{expr}\}\ @\textbf{post}\{\mathit{expr}\} \\
\mathit{advt} & ::= & \textbf{before} \mid \textbf{after} \mid \textbf{around} \\
\mathit{arg} & ::= & t\ \mathit{var} \\
\mathit{expr} & ::= & \textbf{new}\ c \mid \mathit{var} \\
 & & \quad \mid \mathit{expr}.f \mid \mathit{expr}.f = \mathit{expr} \\
 & & \quad \mid \mathit{expr}.m(\mathit{expr}^{*}) \\
 & & \quad \mid \textbf{super}.m(\mathit{expr}^{*}) \\
 & & \quad \mid \textbf{proceed}(\mathit{expr}^{*}) \\
 & & \quad \mid (t)\ \mathit{expr} \\
 & & \quad \mid \textbf{let}\{\mathit{binding}^{*}\}\ \textbf{in}\ \{\mathit{expr}\} \\
 & & \quad \mid \textbf{if}(\mathit{expr})\{\mathit{expr}\}\ \textbf{else}\ \{\mathit{expr}\} \mid \textbf{true} \mid \textbf{false} \\
 & & \quad \mid \textbf{error}(\mathit{expr}) \\
\mathit{binding} & ::= & \mathit{var} = \mathit{expr} \\
\mathit{pcut} & ::= & \mathit{exec}\ \&\&\ \textbf{this}(\mathit{var}) \\
\mathit{exec} & ::= & \textbf{exec}(t\ c.m(\mathit{var}^{*})) \mid \textbf{exec}(t\ c.a(\mathit{var}^{*}) \\
\mathit{prec} & ::= & @\textbf{first} \mid @\textbf{last} \\
\mathit{var} & ::= & \text{a variable name or } \mathit{this} \\
c & ::= & \text{a class name } (\mathit{or}\ \textbf{Object}) \\
f & ::= & \text{a field name} \\
m & ::= & \text{a method name} \\
a & ::= & \text{an advice name} \\
t & ::= & c \mid \textbf{boolean}
\end{array}
$$

Figure 1: Surface syntax

It also is more flexible than AspectJ's aspects, in the sense that the developer regains precise control over the instantiation of aspects. As pointcuts and advice are now regular class members, aspects can extend other aspects as well. As advice are named in ContractAJ, this allows for advice hiding/overriding. In case of overriding semantics, whenever a class with an overriding advice is instantiated, the overriding advice becomes active instead of the overridden advice. This implies that the contracts of an overriding advice should comply with the overridden counterpart, i.e. regular Liskov substitution applies. However, because we prefer to focus on how an advice should comply with the join points in its pointcut, we assume hiding semantics, which do not involve substitution.

Regarding the syntax of pointcuts, shown in the *pcut* rule, there are only two constructs available: Method and advice execution can be captured with `exec`, and the `this` object can be bound. These are the only constructs that we need to implement contract enforcement. However, because enforcement is applied per join point, it is independent of the pointcut language, which is then free to be extended.

Another rule open for extension is the *prec* rule, which shows the syntax of the precedence/composition mechanism. The contract enforcement algorithm only makes use of `@last` to indicate that a particular advice needs to be executed last if there are other advice that apply to the same join point.

Finally, note the optional *c.* in the *method* and *field* rules; it allows for inter-type declarations.

2.2 Semantics

The semantics of ContractAJ is a fairly straightforward extension of ContractJava. It is not a pure extension, in the sense that support for interfaces is removed. The join point model in ContractAJ only includes method and advice executions, as these are the only points in time where we wish to check contracts. In other words, advice can be executed whenever a method, or another advice, is about to be executed.

As mentioned in 2.1, the pointcut language is minimal; it mainly consists of the `exec` construct to capture both method

```
class A {
	@pre x > 0
	public void foo(int x) {...}}
class B extends A {
	@pre x > -10    // Weaker than A's @pre
	public void foo(int x) {...}}
aspect C {
	@pre x > -5 // Also weaker than A's @pre
	around(int x): execution(void A.foo(int))
		&& args(x) {...}}

B inst = new B();
inst.foo(-8); // Contract violation in C
```

Figure 2: Aspect inadvertently causing contract violation

and advice executions. While there is no `args` construct to bind parameters as in AspectJ, method/advice arguments are bound within the `exec` construct itself. For simplicity, we require that all method/advice parameters are bound, as they may be needed when checking contracts. Wild cards are still allowed in the class and method name.

2.2.1 Execution pointcuts and subtyping

An important difference between ContractAJ's `exec` and AspectJ's `execution` construct is the way subtyping is treated: In AspectJ, an `execution` pointcut matches if the type that is specified is a subtype of the join point's *dynamic* type. For example, given that class `B` extends `A` and we execute `B b=new B(); b.foo();` , then the pointcut `execution(* A.foo)` will match on the latter statement. ContractAJ's `exec` construct however only matches if the type specified in the pointcut is equal to the join point's *static* type. This implies that `exec(* A.foo())` would not match, but `exec(* B.foo())` of course would match, disregarding the dynamic type of `b`. At first, we made this choice because it would be more convenient to implement our contract enforcement algorithm.

However, we also found that using this semantics for `exec` is more appropriate from a DbC perspective, which is explained with the AspectJ example in Fig. 2. We assume in this example that aspect `C` and class `B` were written by two different people, unaware of each other's work. Suppose Alice wrote aspect `C` and Bob wrote class `B`. Alice refers to the `A` class in her pointcut, and makes sure that the precondition of her advice is weaker than `A`'s. Bob as well ensures that `B.foo`'s precondition is weaker than `A`'s. Whenever the static type of method calls is `A`, no contract violations will occur, as long as the caller complies with `A`'s preconditions. If it is `B` however, the aspect is applied and it should comply with `B`'s contracts. Alice however was not aware of `B`, resulting in her aspect's precondition failing. This problem is easily avoided by adopting ContractAJ's semantics for `exec`: Use the static type to perform matching.

2.2.2 Lookup semantics

Weaving advice is done by extending the method lookup mechanism. As a result, advice are late-bound, which allows for more flexibility and, in turn, enables ContractAJ to be mapped to a wider range of aspect-oriented languages. Method lookup in ContractAJ follows these steps:

1. Perform normal method lookup, i.e. traverse the subtype hierarchy upwards until the method body is found.
2. For all object instances that belong to a class with pointcuts, try to match these pointcuts. This implies that a pointcut only becomes active as soon as the class contain-

ing the pointcut has been instantiated. The pointcut will remain active until the instance is out-of-scope. If there are multiple instances of the same class, the pointcut is checked multiple times as well.

3. Given all matching pointcuts, the composition mechanism will determine the order in which the corresponding advice should be executed. Store this composition order in a global map, associated with the current join point.
4. Remove the first advice from the composition and execute this advice. If the composition is empty, execute the method body determined by normal method lookup.

A similar set of steps exists for advice execution; the main difference is that there is no analogous concept of normal method lookup in this case, i.e. there is no step 1.

The semantics of a `proceed` call are straight-forward: Retrieve the composition for the join point this advice is being applied to, remove the first element and execute it.

3. The advice substitution principle

The advice substitution principle is the result of adapting the Liskov substitution principle [3] to aspect-oriented languages. This advice substitution principle is not new; the term was first introduced in Wampler [6]. This paper will cover the principle as well, but also takes into account advice execution join points and relaxes the principle at shared join points. The key idea to adapt Liskov substitution to AOP is to view the weaving of an advice as a form of substitution, which it effectively is. This is quite easy to see if all advice are viewed as around advice: This is not a simplification, as a before advice simply is an around advice where the proceed statement at the end is implicit. An after advice similarly is an around advice where the proceed statement in the beginning is implicit. If the pointcut associated with an around advice matches on a certain join point, then that join point essentially is replaced with the execution of that advice. In other words, the join point representing a method call/execution is *substituted* for the execution of an advice.

3.1 Around advice

From the point of view that advice weaving can be seen as a form of substitution, Liskov substitution translates itself as follows to around advice:

Given an around advice that is applied to a particular join point representing an execution of method/advice `y`, where the static type of the receiver is class `X`, then:

- The around advice's preconditions must be equal to or weaker than those of `X.y`.
- The around advice's postconditions must be equal to or stronger than those of `X.y`.
- The around advice must preserve all invariants of `X`.

3.2 Before and after advice

Even though a before/after advice can be interpreted as an around advice, a distinction must be made from a design by contract perspective. This is due to the fact that the postcondition of a before advice intuitively refers to the moment just *before* executing the implicit proceed statement at the end of the advice body. As a consequence, the advice substitution rule for postconditions becomes: The before advice's postconditions may not invalidate the preconditions of `X.y`.

Similarly, an after advice's precondition refers to the moment *after* the implicit proceed statement in the beginning; its rule for preconditions becomes: The after advice's precondition may rely on the postcondition of `X.y`.

Next to these two changes, the other rules concerning around advice remain unaltered for before/after advice.

3.3 Shared join points

If two or more advice need to be applied to the same join point, the above rules still hold. If an aspect `A` has higher precedence than an aspect `B`, it is sufficient that the contracts of `A` comply with those of the static type of the join point. At this point we diverge from the advice substitution principle presented in Wampler [6], which suggests that `A` should also comply with `B`. We argue that it is sufficient to comply the static type's contracts, as the caller will only take those contracts into account. The caller can be oblivious towards aspects, and aspects themselves can be oblivious to the fact that other aspects are sharing join points with them.

3.4 Aspects that intercept advice execution

As our join point model includes the execution of advice, it is possible that one advice is applied to another advice. For example, it is possible that advice `myAdvice` intercepts executions of `myMethod`, and that there is another advice `myMetaAdvice` which intercepts executions of `myAdvice`. In this case the advice substitution rules can be applied just the same; e.g. the preconditions of `myMetaAdvice` may not be stronger than those of `myAdvice`, which may not be stronger than those of `myMethod`.

3.5 Relation to quantification and obliviousness

Finally, it is important to keep in mind that the advice substitution principle is checked per join point. As a pointcut is a quantification mechanism, an advice may apply to a large set of join points, which correspond to various different methods, each with their own contracts. In other words, an advice should comply with the contracts of *all* the different methods it applies to. This number of different methods may be a suitable metric to measure how tightly coupled an aspect is. However, it should be combined with another metric that indicates how closely an aspect interacts with the base system. For example, a logging aspect is often applied to a large number of methods, due to the use wildcards, yet the aspect is unlikely to cause harm, as a logging aspect typically only observes the base system and is only loosely coupled in this sense.

In terms of obliviousness, strictly enforcing the advice substitution principle means that the base system can effectively be unaware of aspects. If a developer makes a method call and ensures its pre- and postconditions, the substitution principle ensures that the postcondition should hold, even in the presence of subtypes and aspects. However, some aspects, such as an authentication aspect, are likely to break the advice substitution principle: For example, if a user is no longer logged in to his internet bank (due to a session time-out), and attempts to access his/her bank account's information, the authentication aspect should deny access. However, if the preconditions in the bank account class do not state "The user must be logged in.", which is realistic in an aspect-oriented implementation, the authentication aspect violates the substitution principle: The precondition was strengthened by the aspect. In such a case, the bank account class cannot be oblivious towards the aspect and should be made aware of it. Further exploration in this direction is however left as future work.

$\boxed{\text{def}^{c}}$

```
$P, c \vdash meth_j \rightharpoonup_{\mathsf{adv}} meth\_check_j$    $P, c \vdash meth_j \rightharpoonup_{\mathsf{pre}} meth\_pre_j$    $for\ j \in [1, m]$
$P, c \vdash meth_j \rightharpoonup_{\mathsf{post}} meth\_post_j$    $P, c \vdash adv_k \rightharpoonup_{\mathsf{adv}} adv\_check_k$    $for\ k \in [1, n]$
--------------------------------------------------------------------------------
$P \vdash$ class $c$ extends $c'$          $\rightharpoonup_{\mathsf{d}}$  $P \vdash$ class $c$ extends $c'$ $\{meth_1 \ldots meth_m\ adv_1 \ldots adv_n\}$
$meth_1 \ldots meth_m\ adv_1 \ldots adv_n$          class $contract\_c$ extends Object {
                                                  $meth\_check_1 \ldots meth\_check_m$
                                                  $c.meth\_pre_1 \ldots c.meth\_pre_m\ c.meth\_post_1 \ldots c.meth\_post_m$
                                                  $adv\_check_1 \ldots adv\_check_n$
                                                }
```

$\boxed{\text{adv}^{m}}$

```
$e_{pre}\mathsf{Pre}_P\ \langle t, md\rangle$    $e_{post}\mathsf{Post}_P\ \langle t, md\rangle$
--------------------------------------------------------------------------------
$P, t \vdash t'\ md(t_1\ x_1,\ t_2\ x_2,\ \ldots,\ t_m\ x_m) \rightharpoonup_{\mathsf{adv}}$
@last around $t'\ md(t\ dyn,\ t_1\ x_1,\ t_2\ x_2,\ \ldots,\ t_m\ x_m)$ :
exec($t'\ t.md(t_1\ x_1,\ t_2\ x_2,\ \ldots,\ t_m\ x_m)$) && this($dyn$) {
  if($e_{pre}$) {
    $dyn.md\_hierarchyPreCheck(x_1,\ x_2,\ \ldots, x_m)$
    let ($returnVal$ = proceed($x_1,\ x_2,\ \ldots, x_m$)) {
      if($e_{post}$) {
        $dyn.md\_hierarchyPostCheck$(true, $x_1,\ x_2,\ \ldots, x_m$)
      } else {
        error("Postcondition violation, blame this")}
      $returnVal$
    }
  } else {
    error("Precondition violation, blame getStackStrace[1]")}
}
```

$\boxed{\text{adv}^{around}}$

```
$e_{pre\_a}\mathsf{Pre}_P\ \langle t, md\rangle$    $e_{post\_a}\mathsf{Post}_P\ \langle t, md\rangle$
$e_{pre\_b}\mathsf{Pre}_P\ \langle t'', md\rangle$    $e_{post\_b}\mathsf{Post}_P\ \langle t'', md\rangle$
--------------------------------------------------------------------------------
$P, t \vdash$ around $t'\ md(t_1\ x_1,\ t_2\ x_2,\ \ldots,\ t_m\ x_m)$ :
exec($t'\ t''.md'(t_1\ x_1,\ t_2\ x_2,\ \ldots,\ t_m\ x_m) \rightharpoonup_{\mathsf{adv}}$
@last around $t'\ md(t\ dyn,\ t_1\ x_1,\ t_2\ x_2,\ \ldots,\ t_m\ x_m)$ :
exec($t'\ t.md(t_1\ x_1,\ t_2\ x_2,\ \ldots,\ t_m\ x_m)$) && this($dyn$) {
  if($e_{pre\_b}$) {
    if(!$e_{pre\_a}$) {error("Substitution principle error: Precondition too strong")}
    let ($returnVal$ = proceed($x_1,\ x_2,\ \ldots, x_m$)) {
      if ($e_{post\_b}$) {
        if(!$e_{post\_a}$) {error("Substitution principle error: Postcondition too weak")}
      } else {
        error("Postcondition violation, blame dyn")}
      $returnVal$
    }
  } else {
    error("Precondition violation, blame getStrackTrace[1]")}
}
```

$\boxed{\text{hier}^{pre}}$

```
--------------------------------------------------------------------------------
$P, c \vdash t\ md(t_1\ x_1,\ t_2\ x_2,\ \ldots,\ t_m\ x_m)\ \{e\}$ @pre{$e_{pre}$} @post{$e_{post}$}
$\rightharpoonup_{\mathsf{pre}}$ boolean $md\_hierarchyPreCheck\ (t_1\ x_1,\ t_2\ x_2,\ \ldots,\ t_m\ x_m)$ {
let ($next = (\exists^{?}$super.$md\_hierarchyPreCheck\ (x_1,\ x_2,\ \ldots,\ x_m)$ || $res = e_{pre}$) {
  if (!$next$ || $res$) {
    $res$
  } else {
    error("Substitution principle error: Precondition too strong")}
}
```

Figure 3: Judgements that implement contract enforcement and blame compilation

4. Contract enforcement in ContractAJ

Based on the contract enforcement algorithm for Contract-Java in Findler et al. [2], contract enforcement for ContractAJ is implemented as a number of judgements. Applying these judgements will transform a ContractAJ program into a version where contract enforcement has been added. Note that, in this transformation, we do not modify any parts of the original program, but only add a number of aspects. The most important judgements of the contract enforcement algorithm are shown in Fig. 3. These judgements make use of two relations on the abstract syntax: Pre_P and Post_P. These are used to retrieve the pre/postconditions of a particular method/advice. Given a pair consisting of the name of a method/advice, and a type, the expression representing the pair's pre/postconditions is returned.

The $[def^{c}]$ rule in Fig. 3 specifies the $\rightharpoonup_{\mathsf{d}}$ judgement, which creates the contract-checking aspect for each class in the original program. Keep in mind that in ContractAJ, aspects are classes; i.e. the use of pointcuts and advice is allowed in classes. As such, the contract-checking aspects are implemented as classes. Suppose the original program contains a class `A`, then its contract-checking aspect is named `contract_A`. For each method in `A`, there is one advice `contract_A` and two methods are added to `A` (using inter-type declarations). The advice, specified in rule $[adv^{m}]$, is applied whenever the corresponding method in `A` is called. The two methods added to `A` are helper methods that check whether Liskov substitution holds. One method checks Liskov substitution on the precondition, which is specified in rule $[hier^{pre}]$; the other checks the postconditions. For each advice in `A`, there is one advice in `contract_A` that checks it. Rule $[adv^{around}]$ specifies this advice in case it checks an around advice. (The cases for before and after advice are not shown, as they are similar.) To avoid confusion, henceforth we will call the advice that implement contract-checking "contract-advice". Normal advice defined in the original program will be called "user-advice".

Rule $[adv^{m}]$ specifies the contract-advice that checks methods. An around contract-advice is used for this purpose, associated with an `exec` pointcut that matches whenever the static type is *t* in a method execution. The pointcut also binds `this` to the `dyn` variable. In the contract-advice body, the precondition of *t* is checked. If it fails, we state that `getStackTrace[1]` is to be blamed. In most cases, this simply means that the caller of the method is to be blamed. If however, one or more user-advice are applied to the method call, the last user-advice in the composition is to be blamed. It can only be the last advice; otherwise an error would have been generated earlier by the contract-advice that check each user-advice. After checking the precondition of *t*, we invoke the Liskov substitution-checking helper method on `dyn`, which traverses up the subtype hierarchy tree to check whether each precondition complies with its supertype, as specified in rule $[hier^{pre}]$. (The $\exists^?$ symbol before `super.md_hierarchyCheck` means: If `md_hierarchyCheck` does not exist in `super`, the call is left out.) After Liskov substitution has passed for the preconditions, we can make a proceed call to execute the method that we are checking. Once this is done, all that is left is to check the postconditions, analogous to the preconditions.

The $[adv^{around}]$ rule specifies the contract-advice that checks around user-advice. Even though the rule looks more complex than the $[adv^{m}]$ rule, it is more of a special case of $[adv^{m}]$, because the functionality needed from $[hier^{pre}]$ has been inlined in $[adv^{around}]$. This was done because advice substitution is simpler than Liskov substitution: We only need to comply with the static type's contracts. We do not need to comply with any advice, because advice cannot be called explicitly and therefore the corresponding aspects will not appear as static types. When examining the $[adv^{around}]$ rule, the similarity with $[adv^{m}]$ becomes clear: The advice's precondition is first checked; we then check whether the precondition is equal or weaker than the static type's (t'') precondition. If these checks pass, we can proceed with executing the user-advice and check the postconditions analogous to the preconditions.

Little changes in case of before and after user-advice: For before advice, we remove the postcondition substitutability check. In case the before advice interferes with the next element in the composition, this will be detected in its precondition check, as it will blame `getStackTrace[1]`. The after advice is treated similarly, we remove the precondition substitutability check. If the normal precondition check fails, the precondition itself is to be blamed. Because this is an after advice, the proceed was already called implicitly and the postcondition of that call must have succeeded to be able to reach this point.

5. Related work

Most work in the area of DbC, related to aspects, is about the use of aspects to implement contract enforcement in object-oriented languages (DbC *by* aspects). However, there are few papers that apply DbC to an aspect-oriented language (DbC *for* aspects): In Zhao et al. [7], Pipa is presented, an aspect-oriented extension to the JML behavioral interface specification language. The language's semantics is only discussed informally and does not mention a notion of advice substitutability. In Lorenz et al. [4], aspects are classified as agnostic, obedient or rebellious. Developers can indicate which of the three types an aspect belongs to, and a prototype implementation called CONA will then perform contract enforcement using this information. Our enforcement of advice substitution is simpler in the sense that it does not involve any classification; the substitution principle only enforces that, when calling a method, the developer does not need to be aware of subtypes nor aspects. The CONA tool uses aspects to enforce contracts on objects, but uses objects to enforce contracts on aspects. Our approach purely uses aspects, and no modifications to the base program are needed. Finally, Agostinho et al. [1] as well is based on the advice substitution principle of Wampler [6], but focuses more on applying the principle to various concrete aspect-oriented languages. Our approach is to implement the principle only on ContractAJ, being a minimal, but flexible aspect-oriented language that leaves some parameters open for extension.

6. Conclusion and future work

This paper has presented a contract enforcement algorithm for the aspect-oriented ContractAJ language, implemented by means of aspects. This language was designed such that it covers a wide range of aspect-oriented languages. The advice substitutability principle, and how it closely relates to Liskov substitution, was discussed in detail as well. Future work includes implementing the algorithm in, for example, AspectJ and adapting the contract soundness proof of Findler et al. [2], to show that our contract enforcement algorithm always finds contract violations if they occur. Another interesting direction to take is to be able to statically find violations of advice substitutability. If no such violations are found, the base system can effectively be oblivious towards advice. If violations cannot be avoided, perhaps the base system should somehow become aware of the interaction that is occurring, which is an interesting direction of its own.

References

[1] Sérgio Agostinho, Ana Moreira, and Pedro Guerreiro. Contracts for aspect-oriented design. In *Proceedings of the 2008 AOSD workshop on Software engineering properties of languages and aspect technologies*, SPLAT '08, page 1:1–1:6, New York, NY, USA, 2008. ACM.

[2] Robert Bruce Findler and Matthias Felleisen. Contract soundness for object-oriented languages. In *Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '01, page 1–15, New York, NY, USA, 2001. ACM.

[3] Barbara H Liskov and Jeannette M Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, November 1994.

[4] David H Lorenz and Therapon Skotiniotis. Extending design by contract for Aspect-Oriented programming. *Order - A Journal On The Theory Of Ordered Sets And Its Applications*, 2005.

[5] H. Rajan and K.J. Sullivan. Classpects: unifying aspect- and object-oriented language design. In *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, pages 59 – 68, May 2005.

[6] D. Wampler. Aspect-oriented design principles: Lessons from object-oriented design. In *Sixth International Conference on Aspect-Oriented Software Development (AOSD'07), Vancouver, British Columbia*, 2007.

[7] Jianjun Zhao and Martin Rinard. Pipa: A behavioral interface specification language for aspect. In Mauro Pezzè, editor, *Fundamental Approaches to Software Engineering*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.