Contracts and Aspects

David H. Lorenz and Therapon Skotiniotis

Technical Report NU-CCIS-03-13
College of Computer and Information Science
Northeastern University
360 Huntington Avenue 161 CN
Boston, Massachusetts 02115 USA
{lorenz,skotthe}@ccs.neu.edu

Abstract. Recently it has been shown that Design by Contract (DBC) in OOP have been erroneously enforced by existing DBC technologies. The crosscutting nature of DBC and its runtime enforcement, make it an ideal problem for AOSD to solve. In this paper we present an AOSD DBC tool, `conaj` that provides DBC support by means of aspects, accommodating also pre- and post-conditions for advice. `conaj` is based on the analysis of the relationships of pre- and post-conditions between aspects and their advised objects. Assertions on advice guarantees that addition of aspects honors the underlying systems obligations and does not break existing system behavior due to erroneous advice attachments or conflicting aspect–object obligations. AOSD development benefits as runtime error reporting becomes more accurate and advice expectations and obligations are now explicitly defined and validated. As a result increasing the ability to reason about composition as well as the behavior of aspects.

1 Introduction

The use of *assertions* to verify program correctness dates back to the basics of programming [7, 8]. Having their roots in theory and verification, assertions have made their way to procedural [22], functional [5] as well as Object-Oriented Programming (OOP) languages [18]. The Eiffel [18, 19] OOP language, for example, provides the definition of assertions as part of the language. This discipline of programming is generally called: *Design by Contract* (DBC).

DBC makes the obligation–benefit contract between software consumers and providers explicit. Each instance method defines the valid states in which its execution can start (*precondition*), and the states in which it will terminate (*postcondition*). A more general assertion (*invariant*), which is maintained before as well as after any externally observable state of an object, ensures that the object maintains an acceptable state throughout the program's execution. Invariants are particularly useful as an inductive hypothesis: What ever is assumed should be inductively provable.

Checking assertions for methods is an important technique which improves software stability and reliability [18]. Many languages provide package support for DBC, e.g., Java [1, 12, 9, 13], Ada [17], C [22]. In Eiffel [19], Sather [20], and Blue [11] the provision of DBC is an integral mechanism of the language itself.

1.1 Contribution

Contract checking is a crosscutting concern. Contracts are systematically placed throughout the code, and their evaluation also crosscuts the program's execution at systematically, well defined, execution points [16, 25, 2]. This paper presents the use of aspects to enforce contracts and the use of contracts to check advice.

The use of aspects for contracts has several advantages. First, aspects can bring DBC to languages that currently have no support for contracts. We present an AODBC tool that implements this approach in AspectJ. The deployment of DBC in `conaj` is a separate aspect inside a program, which developers activate or deactivate at will. Furthermore, all the code that deals with contracts (and their enforcement) is separated from the application's code, both at the source code level and at the binary level, allowing for more reuse and easier integration with third-party software.

Second, aspects can provide a supplemental implementation of DBC in a language that already has some support for DBC. For example, DBC in OO languages has an interesting interplay with inheritance [4, 6], and more specifically with behavioral subtypes [21, 15]. Findler and Felleisen [4] point out a common error in the implementation of contract checking in OO languages: computing the disjunctions of preconditions of overridden methods does not capture the cases where subtypes violate the law of type substitutability [15]. In [4] the authors further provide the semantics of a correct contract checking mechanism, and prove its soundness using Java's core semantics. This papers builds on their results.

Third, this paper discusses the usage of pre- and post-conditions for advice. We define what program states are being specified by pre- and post-conditions in advice, and how these conditions should be checked at runtime. The dependency between advice obligations and the base program's obligations are defined in a form that maintains the original program's semantics and disallow aspects from breaking them. The paper concludes by presenting a set of judgments that define the compilation steps required to extend `conaj` and provide runtime contract checking for both methods and advice.

2 Aspects for Contracts

The key idea in using aspects for contracts is to provide a solution that encapsulates both the code that specifies the contracts and the code that implements the contract checking mechanism that is to enforce proper subtype behavior as defined by [4]. While several contract checking tools have explored alternatives other than AOP [12, 9, 1, 19], an AOSD implementation is better suited for DBC. A requirement that we imposed on the aspectual design of `conaj` was that code that deals with contracts is solely handled by aspects. Given an OO program along with contract annotations for each type, the solution should only non-invasively introduce new "components" as aspects and keep the original classes unaffected[1].

[1] By "unaffected" we also refer to the absence of *introductions* as known in AspectJ

2.1 Architecture

`conaj` (Figure 1) is a preprocessing tool, implemented using DemeterJ.[2] The grammar for `conaj` is specified by a DemeterJ class file (`conaj.cd`), while all operations for analysis and output are specified inside DemeterJ behavior files (`.beh`). Compiling these files with DemeterJ produces all binary files (`.class` files) that make up `conaj`.

`conaj` takes as input a list of files with extension `.conaj` or `.java`, containing code written in a superset of Java which accommodates for contract definitions. The input files are parsed and analyzed by `conaj` producing two sets of files as output:

1. Pure Java `.java` files (one for each `.conaj` file), where all contract related code (if present) is commented out.
2. AspectJ `.aj` files (one for each input *type* that holds contract definitions) with appropriate pointcuts, advice, and methods for enforcing contracts.

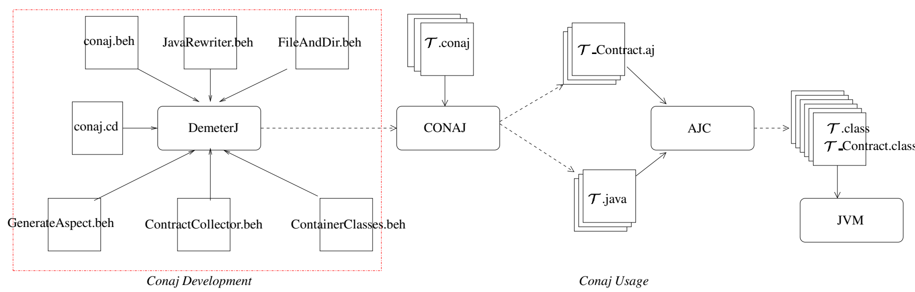

Fig. 1. `conaj` Architecture

2.2 Syntax

`conaj` is a preprocessor, implemented in an adaptive style in DemeterJ. `conaj` introduces four new keywords to Java's set of keywords, **`@pre`**, **`@post`**, **`@invariant`**, and **`old`**. The specification of `conaj`'s grammar (Figure 2) uses DemeterJ's [27] syntax for class files. Class names denoted by '[ ]' are optional. Rules with '=' denote concrete classes and their members and ':' denotes inheritance from abstract classes.

Definitions of **`@pre`** and **`@post`** conditions are allowed anywhere in the body of a method, and also after the signature of a method in an interface definition. **`@invariant`** definitions are allowed at the beginning or at the end of a class or an interface definition. Any valid Java boolean expression e_b can be placed inside an invariant, pre- or post-condition definitions. In the case of a post-condition, the special keyword **`old`** can be used to refer to the object state before a method's execution. The usage of a method's name (m) inside it's *PostCondition* block is used to refer to the returned value

[2] `conaj` was developed with DemeterJ version 0.8.6 [14, 27] and AspectJ [26] version 1.0.6

```
                 /* Extensions for Conaj */
PreCondition   = @pre{ ConditionalOrExpression } .
PostCondition  = @post{ ConditionalOrExpression } .
Invariant      = @invariant{ ConditionalOrExpression } .
ClassBody      = { [InvHeadTail] ClassBodyDecls [InvHeadTail] } .
InvHeadTail    = Invariant.
InterfaceDecl  = interface Identifier [ extends NameList ]
                   { [Invariant]InterfaceMemberDecls [ Invariant ] }.
MethodDecl     = MethodModifiers MethodSignature AnyBlock .
AnyBlock       = A_Block | A_SemiColon .
A_Block        = { [ BlockHead ] BlockStatements
                   [ BlockTail ] } .
BlockHeadTail  : BlockTail | BlockHead
                   common [ PreCondition ] [ PostCondition ] .
BlockHead      = .
BlockTail      = .
A_SemiColon    = ; [ PreCondition ] [ PostCondition ] .
                 /* Grammar for the Java Language. (omitted) */
```

Fig. 2. `conaj` Grammar (using DemeterJ Class Graph syntax)

obtained (if the method returns a value other than `void`) after the method m completes its execution.

2.3 Code Generation

`conaj` generates an aspect class for each Java type that uses contracts. These pointcuts are used in enforcing pre- and post-conditions. In the case of an invariant definition the generated pointcut captures all public method calls made to an object. The invariant pointcut is then advised with a before and an after advice, both of which call a method inside the aspect that checks the invariant condition.

In the case of methods with contract definitions, the aspect defines two pointcuts: one to capture method calls and another to capture method execution joinpoints. Both calls and executions of the method *must* be caught in order to deal with the cases where the dynamic type of an instance is different from its static type.

A before advice is generated if a pre-condition is specified, and a post advice if a post-condition is specified. The before advice first tests that the pre-condition expression evaluates to true and then calls the checks for the pre-condition hierarchy. The pre-condition hierarchy tests that the implication $supertype_{pre} \rightarrow subtype_{pre}$ holds for the whole chain of supertypes from the receiver's static type and up including interfaces. Both checks should hold. If any of the checks fails an appropriate exception is thrown. Checking post-conditions is done in a similar fashion with only a change in the logical implication that is checked for the hierarchy of supertypes $subtype_{post} \rightarrow supertype_{post}$.

A second advice is generated to check executions of the method. The checks performed are essentially the same as for method calls, except this time the hierarchy check (both for pre- and post-conditions) starts from the dynamic type of the receiver instead of its static type. The synergy of these two pointcuts allows the correct sequence of contract checks in the situations where the static and dynamic type of the receiver instance differ.

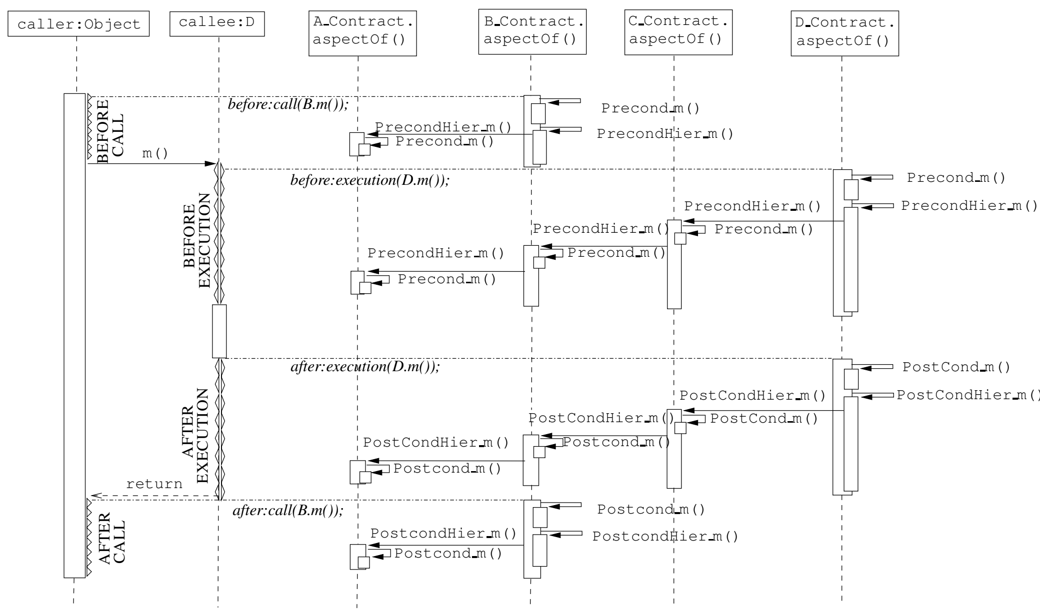

Fig. 3. Interaction diagram for the code fragment { `B b = new D(); b.m();` }

Figure 3 depicts more concretely the interactions between aspects that implement the proper contract checking for the following scenario. Consider a program that consists of four classes, A, B, C and D with an inheritance relation $A <: B <: C <: D$ where $<:$ denotes inheritance, i.e., $supertype <: subtype$. The method m is defined in A and is overridden in each subclass of A. During program execution, a call to m is made from a `caller` of type `Object`. The static type of the receiver is B while its dynamic type is D. In Figure 3 we denote instances of aspects by using their name and the AspectJ call that returns an aspect instance (i.e. `aspectOf()`).

First, the contract obligations for the static type of the receiver are checked. On success of the static types contract obligations, the contract obligations for the *dynamic* type are checked. On success, m is executed. The opposite sequence of interactions is performed for the symmetrical case of post-conditions on m.

2.4 Aspect generated for the Stack Example

Listing 1.1 shows an example of a Stack implemented in `conaj`, and Listing 1.2 the generated aspect. In Listing 1.2 the aspect definition is declared as `privileged` to allow access to the instance's private data members if needed. In order to support access to the instance's old state, we introduce an implementation of `clone()`. This is the only situation where we relax our constraint of *not* using AspectJ's introductions. The extra pointcut `scope` is used to exclude calls to methods that are initiated by the aspect.

Listing 1.1. The Stack example in `conaj`

```
class MyStack {
 @invariant{0<=size() && size()<=maxSize}
 protected Vector elements;
 protected int maxSize;
 protected int size;

 MyStack(int n){
  elements = new Vector(n);
  maxSize = n;
  size = 0;
 }

 public int size() { return size; }

 public void push(int i){
  @pre{!full()}
  @post{!empty() && top()==i && size()==old.size() + 1}
  Integer val = new Integer(i);
  elements.add(val);
  size=elements.size();
 }

 public int pop(){
  @pre{!empty()}
  @post{!full() && size()==old.size() - 1}
  Integer result = (Integer) elements.lastElement();
  int index = elements.lastIndexOf(result);
  elements.remove(index);
  size = elements.size();
  return result.intValue();
 }

 public int top(){
  @post{size() == old.size()}
  Integer result = (Integer) elements.lastElement();
  return result.intValue();
```

```
  }

  public boolean full(){return (size() == this.maxSize);}

  public boolean empty(){return elements.isEmpty();}
}
```

Listing 1.2. Generated aspect for `MyStack`. Showing relevant code for invariant and method `push()`

```
privileged aspect MyStack_Contract {
  declare parents : MyStack implements Cloneable;
  public Object MyStack.clone(){
    try {
      return super.clone();
    } catch (Exception e){ //Error }
  }

  MyStack old;
  pointcut scope():
    !within(MyStack_Contract)
    && !cflow(withincode(* MyStack_Contract.*(..)));
  pointcut MyStack_push(MyStack trg_instance, int i):
    call( public * MyStack.push(..))
    && !call(public * (MyStack+ && !MyStack).push(..))
    && args(i) && target(trg_instance) && scope();
  pointcut dynamic_MyStack_push(MyStack trg_instance, int i):
    execution(public * MyStack.push(..))
    && !execution( public * (MyStack+ && !MyStack).push(..))
    && args(i) && target(trg_instance) && scope();
  // PCD for Invariant
  pointcut MyStack_invariant(MyStack trg_instance):
    call(public * MyStack.*(..))
    && !call(public * (MyStack+ && !MyStack).*(..))
    && target(trg_instance) && scope();

  before(MyStack trg_instance):
    MyStack_invariant(trg_instance){
      if (!checkInvariant(trg_instance)){ //Invariant Error }
    }
  before(MyStack trg_instance , int i ):
    MyStack_push( trg_instance , i ){
      old = (MyStack) trg_instance.clone();
      boolean res = PreCond_push( trg_instance , i);
      boolean next = true ;
      if (!res){ //PreCond Error }
      if (!next) { //HierPreCond Error }
    }
  before(MyStack trg_instance , int i ):
    dynamic_MyStack_push( trg_instance, i ){
```

```
    boolean hierResult = (true);
    boolean res = PreCond_push( trg_instance , i);
    if (hierResult && ! res){ //HierPreCond Error }
  }
after(MyStack trg_instance , int i ):
  MyStack_push( trg_instance, i ) {
    boolean res = PostCond_push( trg_instance, i);
    if(!res){ //PostCond Error }
    boolean postHier = PostCondHier_push(trg_instance, res,i);
    if(!postHier){ //HierPostCond Error }
  }
after(MyStack trg_instance , int i ):
  cast_MyStack_push( trg_instance, i ){
    boolean postResult = PostCond_push( trg_instance, i);
    if(!postResult){ //PostCond Error }
    boolean postHier = (true);
    if(!postHier){ //HierPostCond Error }
  }
after(MyStack trg_instance):
  MyStack_invariant(trg_instance){
    if (!checkInvariant(trg_instance)){ //Invariant Error }
  }
public boolean checkInvariant(MyStack trg_instance){
  if (0<=trg_instance.size()&&trg_instance.size()<=trg_instance
      .maxSize)
    return true;
   else
    return false;
}
public boolean PreCond_push(MyStack trg_instance,int i){
  if(!trg_instance.full())
    return true;
  else
    return false;
}
public boolean PostCond_push(MyStack trg_instance,int i){
  if( ! trg_instance.empty() && trg_instance.top() == i &&
      trg_instance.size() == old.size() + 1 )
    return true;
  else
    return false;
}
public boolean PreCondHier_push(MyStack trg_instance,int i){
  boolean myPre = PreCond_push( trg_instance , i);
  boolean hierarchy = (true);
  if (!hierarchy || myPre)
    return myPre;
  else
    return false;
}
```

```
public boolean PostCondHier_push(MyStack trg_instance, boolean
    last,int i){
  boolean postResult = PostCond_push( trg_instance, i);
  if (!last || postResult)
   return (true);
  else
   return false;
 }
}
```

3 The Anatomy of an Aspect Implementing a Contract

This section provides the details of `conaj`'s aspect generation. Readers not interested in the details of the generation process may wish to skip to the next section.

We describe the general structure of an aspect that implements a contract obligation for a Java type. The generalization is based on AspectJ code and terms denoted as $\langle$TERM$\rangle$ which denote type specific information. The meta variables (Table 1) are place

Table 1. Meta variables used and their meaning within `conaj`

Terms	*Definition*
$\langle$Type$\rangle$	Refers to any Java or user defined type name The special variable $\mathcal{T}$ refers to the type currently being processed (e.g `MyStack`)
$\langle$Method$\rangle$	Refers to a type's method name (e.g. `push`). An index is used when a task is repeated for each of the type's methods
$\langle$SuperType$\rangle$	Refers to a *direct* super type name of $\mathcal{T}$. An index is used to iterate through all super types.
$\langle$Invariant code$\rangle$	Refers to the segment of Java code provided as the invariant
$\langle \text{Method}_i$ Precondition code$\rangle$	Refers to the segment of Java code provided as the precondition to $\langle \text{Method}_i \rangle$
$\langle \text{Method}_i$ Postcondition code$\rangle$	Refers to the segment of Java code provided as the post-condition to $\langle \text{Method}_i \rangle$
$\langle \mathcal{FA} \rangle$	Refers to a method's list of *formal parameters* (e.g $\langle \text{Type}_1 \rangle \langle \text{Arg}_1 \rangle \ldots \langle \text{Type}_n \rangle \langle \text{Arg}_n \rangle$)
$\langle \mathcal{A} \rangle$	Refers to a method's list of argument names (e.g $\langle \text{Arg}_1 \rangle \ldots \langle \text{Arg}_n \rangle$)
$\langle$Arg$\rangle$	Refers to a name binding for a method/pointcut argument

holders that will be filled in by `conaj` depending on the type $\mathcal{T}$. This provides a flavor of a template which the generator instantiates for each type.

Listing 1.3 gives the first seven lines of the aspect template. The naming convention used is based on name mangling. For a type $\mathcal{T}$, an aspect with the name $\mathcal{T}$_`Contract` is defined. This definition will reside in a file named after the aspect name with an extension `.aj`. Lines 2–6 define a method `clone()` for $\mathcal{T}$. The introduction adds the interface `Cloneable` to the list of implemented interfaces and also provides a definition for the method `clone()`. Line 8 provides an aspect member definition to hold a reference to the cloned old copy of $\mathcal{T}$. The pointcut definition `scope()` is defined so that we can exclude calls to methods of $\mathcal{T}$ that occur inside the contracts code. This is necessary to prevent an infinite sequence of method-advice execution that leads to non-termination.

Listing 1.3. Aspect definition and the introduction of `clone`

```
1  privileged aspect ⟨𝒯⟩_Contract {
2
3    declare parents : ⟨𝒯⟩ implements Cloneable;
4    public Object ⟨𝒯⟩.clone(){
5      //Implementation of "shallow" cloning for usage with old
6    }
7    //Reference previous state of object
8     ⟨𝒯⟩ old;
9
10   pointcut scope(): !within(⟨𝒯⟩_Contract)
11     && !cflow(withincode(* ⟨𝒯⟩_Contract.*(..)));
```

Listing 1.4 gives the two pointcuts that are defined for each method $\langle \text{Method}_i \rangle$ of $\mathcal{T}$. The first pointcut definition (lines 30–33) is named by concatenating the type's name, an underscore and the method's name. The pointcut's arguments are made up of the $\langle \text{Method}_i \rangle$'s formal arguments, as they appear in the method's signature. In order to be able to expose the running instance of $\mathcal{T}$, we include an extra argument `tInst` as an argument to the pointcut. Inside the pointcut body, lines 31–32, is an AspectJ idiom [23] that captures all calls made *exclusively* to type $\mathcal{T}$. By 'exclusively' we mean those calls that are made on instances whose static type is $\mathcal{T}$ and not any calls to any subtypes of $\mathcal{T}$. Using AspectJ's `args` and `target` primitive pointcuts, arguments and the running instance of $\mathcal{T}$ are bound to the pointcut's arguments. These bindings will be used later on in the evaluation of code defined in the type's contract.

The second pointcut definition, lines 35–38, deals with situations where the running instance of an object is of type $\mathcal{T}$ but it's static type is not $\mathcal{T}$. The body of the pointcut uses another idiom (lines 36–37) which is similar to the idiom used in the previous pointcut deals with `execution` rather than `call` joinpoints. Here the idiom captures *executions* of this type exclusively. This idiom capture executions of type $\mathcal{T}$, and only $\mathcal{T}$, that originated by an invocation on some static type *other* than $\mathcal{T}$. The usage of `args` and `target` are used again to bind the appropriate values to the pointcuts arguments. The template code in Listing 1.4 is instantiated for each method $Method_I$ in $\mathcal{T}$.

Listing 1.4. Pointcut definitions for each method

```
29 // for each Methodᵢ defined in 𝒯 with either a @pre or @post
30  pointcut ⟨𝒯⟩_⟨Methodᵢ⟩(⟨𝒯⟩ tInst,⟨𝓕𝓐⟩):
```

```
31    call( public * ⟨T⟩.⟨Method_i⟩(..))
32    && !call(public * (⟨T⟩+ && !⟨T⟩).⟨Method_i⟩(..))
33    && args(⟨A⟩) && target(tInst) && scope();
34
35  pointcut dynamic_⟨T⟩_⟨Method_i⟩(⟨T⟩ tInst,⟨FA⟩):
36    execution(public * ⟨T⟩.⟨Method_i⟩(..))
37    && !execution( public * (⟨T⟩+ && !⟨T⟩).⟨Method_i⟩(..))
38    && args(⟨A⟩) && target(tInst) && scope();
39  // similar pointcut definitions for each of the remaining methods of T (ommitted)
```

Listing 1.5 captures all calls to public methods made to type $\mathcal{T}$ exclusively. The same idiom is used as the one found in lines 31–32, although this time the pointcut captures any public method's name through the usage of the `*` pattern. In the case of invariants only the running instance is exposed through the pointcut definition.

Listing 1.5. Capturing all public calls, to be advised with invariant code

```
//Invariant PCD
pointcut ⟨T⟩_invariant(⟨T⟩ tInst):
  call(public * ⟨T⟩.*(..))
  && !call(public * (⟨T⟩+ && !⟨T⟩).*(..))
  && target(tInst) && scope();
```

Listings 1.3, 1.4 and 1.5 make up all the pointcuts that an aspect (implementing a contract definition) will contain.

Listing 1.6 shows all before advice (one for each pointcut). Before advice *must be* grouped together in the aspect definition file. The reason for this is the fact that there are situations where a join point will be caught by two different pointcuts. An example is the case where a method has both pre- and post-conditions and the object also has an invariant definition. In this case, before and after advice for the method will take care of pre- and post-conditions respectively. At the same time, the invariant pointcut will add its own before and after in order to check for the invariant condition. Interleaving before and after advice definitions of different pointcuts causes the AspectJ compiler to throw a compile time error[3].

Listing 1.6. Before advice definitions

```
before(⟨T⟩ tInst):
  ⟨T⟩_invariant(tInst){
    if (!Invariant(tInst)){
      // throw Invariant Error exception
    }
  }

// repeat the following 2 before advice for each pointcut
before(⟨T⟩ tInst,⟨FA⟩):
  ⟨T⟩_⟨Method_i⟩(tInst,⟨A⟩){
    old = (⟨T⟩) tInst.clone();
```

[3] ajc gives: `circularity in advice precedence applied to ...`

```
96       res=Precond_⟨Method_i⟩(tInst,⟨A⟩);
97       next=PrecondHier_⟨Method_i⟩(tInst,⟨A⟩) ;
98       if (!res){
99         // throw PreCond Error exception⟨Method_i⟩
100      }
101      if (!next) {
102        // throw HierPreCond Error exception⟨Method_i⟩
103      }
104    }
105  before(⟨T⟩ tInst,⟨FA⟩):
106    dynamic_⟨T⟩_⟨Method_i⟩(tInst,⟨A⟩){
107      hierResult=PrecondHier_⟨Method_i⟩(tInst,⟨A⟩);
108      res=Precond_⟨Method_i⟩(tInst,⟨A⟩);
109      if (!(!hierResult || res)){
110        // throw HierPreCond Error exception
111      }
112    }
```

A before advice that deals with the objects invariant is essentially a call to a method inside the aspect. The `checkInvariant()` method checks the user's invariant code, throwing a runtime exception if it evaluates to `false`. There are two more types of before advice defined. The first before advice, advises calls made to a method $\langle \text{Method}_i \rangle$ of the type $\mathcal{T}$ (lines 93–104). If needed, a copy of the object is created for usage in post-conditions (`old`). Evaluation of the method's precondition code is done by calling a method defined inside the aspect (`PreCond_`$\langle \text{Method}_i \rangle$`()`). Then, according to the appropriate operational semantics [4], the class hierarchy and its preconditions needs to be checked. If there is a hierarchy of user defined classes, then for each of this object's immediate supertypes, the method `PreCondHier_`$\langle \text{Method}_i \rangle$`()` defined in $\langle \text{SuperType}_i \rangle$`_Contract` is called, and their results are combined in a disjunction. The method (recursively) checks that all supertype's pre-condition logically implies the subtype's precondition. If the set of supertypes is empty then the hierarchical check is set to `true`. Finally we check that the method's precondition implies the method's hierarchical precondition result.

The second type of before advice, advises executions of a method $\langle \text{Method}_i \rangle$ due to a call to a static reference of some type $\mathcal{T}'$ and with a dynamic type $\mathcal{T}$ (lines 105–112). The code checks (using the same methods as those given above) for the method's precondition and that the hierarchy is well formed.

Listing 1.7. After Advice

```
// repeat the following 2 after advice for each pointcut
after(⟨T⟩ tInst,⟨FA⟩) returning(⟨Type⟩ result):
⟨T⟩_⟨Method_i⟩(tInst,⟨A⟩){
  res=Postcond_⟨Method_i⟩(tInst,⟨A⟩);
  if(!res){
    // throw PostCond Error exception
  }
  postHier=PostcondHier_⟨Method_i⟩(tInst,res,result,⟨A⟩);
  if(!postHier) {
```

```
133         //throw HierPostCond Error exception
134       }
135     }
136
137   after(⟨𝒯⟩ tInst,⟨ℱ𝒜⟩) returning(⟨Type⟩ result):
138     dynamic_⟨𝒯⟩_⟨Method_i⟩(tInst,⟨𝒜⟩){
139       postResult=Postcond_⟨Method_i⟩(tInst,result,⟨𝒜⟩);
140       if(!postResult){
141         //throw PostCond Error exception
142       }
143       postHier=PostcondHier_⟨Method_i⟩(tInst,postResult,result,⟨𝒜⟩);
144       if(!postHier){
145         // throw HierPostCond Error exception
146       }
147     }
148   after(⟨𝒯⟩ tInst):
149     ⟨𝒯⟩_invariant(tInst){
150       if (!Invariant(tInst)){
151         // throw Invariant Error exception
152       }
153     }
```

Symmetrically there are three types of after advice. After advice deals with post-conditions and with the invariant condition after a method has completed its execution. Listing 1.7 shows the three types of after advice that are generated.

Lines 148–153 checks the invariant condition for type $\mathcal{T}$ by calling the appropriate method inside the aspect. The first after advice in Listing 1.7 deals with calls that are made to the type $\mathcal{T}$. In the cases where the method has a return type, this is captured by AspectJ's `returning` statement used with after advice. First the method's post condition is checked by calling another method defined within the aspect which holds the post-condition code. After that we need to check the class hierarchy and verify that is well formed. A well formed hierarchy for post conditions semantically means that each of the subtypes post-condition should imply its supertypes post-condition [4]. Inside the aspect this is done recursively by the calls to the `PostCondHier_`⟨Method$_i$⟩`()`.

The last type of after advice (lines 137–147) performs the same checks (method post-condition and class hierarchy check), but this is called in situations where a call was made to some static type $\mathcal{T}'$ while the dynamic type of the receiver object was of type $\mathcal{T}$.

Listing 1.8. Aspect Methods

```

 public boolean Invariant(⟨𝒯⟩ tInst){
   (⟨Invariant code⟩)? return true: return false;}

//for each method with a precondition definition
 public boolean Precond_⟨Method_i⟩(⟨𝒯⟩ tInst,⟨ℱ𝒜⟩){
   (⟨Method_i Precodintion code⟩)? return true : return false;}

```

```
218  //for each method with a postcondition definition
219  public boolean Postcond_⟨Method_i⟩(⟨T⟩ tInst,⟨Type⟩ result,⟨FA⟩){
220    (⟨Method_i Postcondition code⟩)? return true : return false;}
221
222  //for each method with a precondition definition
223  public boolean PrecondHier_⟨Method_i⟩(⟨T⟩ tInst,⟨FA⟩){
224    myPre = Precond_⟨Method_i⟩( tInst,⟨A⟩);
225    hierarchy=
226    ⟨SuperType_1⟩_Contract.aspectOf().PrecondHier_⟨Method_i⟩(tInst,⟨A⟩)
227      ||...
228      ||
229      ⟨SuperType_k⟩_Contract.aspectOf().PrecondHier_⟨Method_i⟩(tInst,
           ⟨A⟩);
230    (!hierarchy || myPre) ? return myPre : return false;
231  }
232
233  //for each method with a postcondition definition
234  public boolean PostcondHier_⟨Method_i⟩(⟨T⟩ tInst,boolean last,
         ⟨Type⟩ result,⟨FA⟩){
235    myPost=Postcond_⟨Method_i⟩(tInst,result,⟨A⟩);
236      if (!last || myPost) { // last => myPost
237    return
238        (⟨SuperType_1⟩_Contract.aspectOf().PostcondHier_⟨Method_i⟩(tInst
             ,myPost,result,⟨A⟩)
239        && ...
240        &&
241        ⟨SuperType_k⟩_Contract.aspectOf().PostcondHier_⟨Method_i⟩(tInst,
             myPost,result,⟨A⟩);
242      }else
243        return false;
244  }
```

Listing 1.8 shows the methods inside the aspect that get generated in order to check the pre, post and invariant code given to a method/class (lines 211–220). The hierarchy checking methods (lines 223 to 244) recursively traverse the hierarchy chain and ensure that the appropriate implications hold for pre- and post-conditions. In order to access methods in aspects defined for superclasses, the usage of AspectJ's (`aspectOf()`) is used to refer to an aspect's instance.

`conaj`, the Aftermath The authors in [4] present Contract Java through a set of judgments. Programs are written in a superset of Java and translated to pure Java with extra classes and wrapper methods. Using the same names for the judgments found in [4] the mapping to aspects is straight forward. **[defn^c]** and **[defn^i]** are used to generate the necessary classes along with methods that hold pre, post and invariant definitions for a type. The same operations are present in `conaj` with the difference that aspects are created along with methods inside the aspects to hold pre, post and invariant code for a type. The judgment **[wrap]** takes each Java type declaration with methods that hold pre and post conditions, and a wrapper method is used to call the classes responsible for checking the class hierarchy. The wrapper method's body corresponds to the code found in the advice generated by `conaj`. The hierarchy checks are performed by calling the appropriate method is the aspect bound to an instance supertype.

Developing `conaj` certain decisions concerning the solution's design, but also its final implementation, brought some interesting issues. At first finding the appropriate pointcut that would enable the capture of calls to a type $\mathcal{T}$ and to none of the subtypes of $\mathcal{T}$ was problematic. The solution came about as an idiom which performs as expected, but cannot be inferred by the documentation of AspectJ at the time.

The design could have been more robust, if the ability to dispatch on aspect method names was possible [3]. Currently, `conaj` keeps track of aspect names and type names and inserts the appropriate calls in order to obtain the correct aspect instances.

Still, some AspectJ features have helped the development of `conaj`. The extra reflective power provided through AspectJ's library allows the inspection of more information than with plain Java. For example, the context of a message send (i.e. caller type, receiver static/dynamic type, method call signature). Information that assisted both in error reporting but also acquiring with greater ease information about the running program.

4 Contracts for Advice

One of the hardest part of writing down an aspect definition in order to capture a cross-cutting concern lies in the precise definition of its pointcut(s). Defining not only the correct joinpoints in a program, but also making sure that the aspect does not interfere with some other parts of the program. It is even harder to anticipate an aspect's behavior when the system is extended with new classes.

Reasoning about AOSD programs, and more specifically, with the correctness of an aspect inside a system boils down to "compile, run and see the results". Making pointcuts generic allows for greater possibilities for reuse of the aspect. At the same time however, a generic pointcut may catch more information than what was meant ([25], Chapter 5). Development tools [24] help in the understanding of the resulting program, but cannot verify whether aspects are being used according to the developer's intent.

As an example consider the simple case of a logging aspect on a class A, reporting on calls to accessor and mutator methods (Figure 4). Suppose a new class B is introduced as a subclass of A, which performs logging inside the definitions of its methods. Running the above system will produce two types of logging information for methods in class B. Logger takes effect over both A's and B's method calls in all situations where B is used. The overall effects due to the addition of B are only observable after running the program. In bigger systems, with more complex pointcuts, the effects become hard to understand, as a result, debugging becomes difficult. In order to attach Logger only on A, the AspectJ manual (Chapter 4) proposes an idiom consisting of a runtime check on the target's dynamic type inside the pointcut definition. A better solution would be to

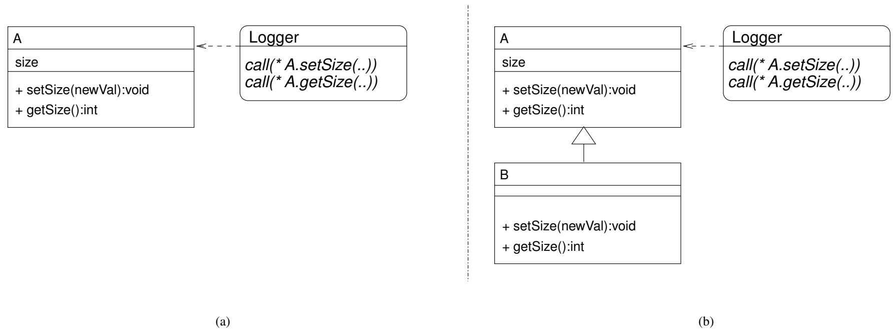

Fig. 4. Logging Example (a) for one class A.Adding B (b) and overriding A's methods toinclude logging inside the methods of B.

express, in the definition of the advice, the acceptable states that the advice can be executed in. In this way, the expected advice behavior is specified and through a contract checking mechanism, this behavior can be verified at runtime. Failing to meet a contract obligation will generate appropriate error messages providing more information about the source of the problem. Reasoning about what went wrong, where it went wrong and why, will be easily pointed out, leading to faster error detection and repair.

4.1 Externally Visible States with Before and After Advice

Pre- and post-conditions in OO programs allow for the specification of *externally* visible states of an instance. These specifications denote not only the set of acceptable states a method can start and finish in, but also play a role in the type relationships found in an OO program. To address the idea of DBC for advice a clear and precise definition of what is considered an *externally* visible states of the aspect is required. Also the interplay between pre- and post-conditions found on advice and pre- and post-conditions found in object instances need to uphold certain restrictions. Specifically, pieces of advice should not alter the program states in a way that will cause a valid method call to no longer uphold it's contract obligations.

We will consider `before` and `after` advice for aspects in AspectJ to investigate (1) what are the externally visible states of instances of Aspects (2) when is the definition of externally visible states of object instances is still valid (3) the interplay of advice pre- and post-conditions with pre- and post-conditions found in object instances.

The following scenario is considered. Having a method m of some type $\mathcal{T}$ in a program P, we advice m through the addition of a new aspect α. In the resulting program P' we can identify some new program states.

The external behavior of an aspect's advice is the same as a call to some method x in P'.[4] Treating advice as some special form of a method (x) found only in P', the externally observable states for an advice are the points in the execution path of P' immediately before a call to x and after completion of x. Based on the above observation

[4] Looking at AspectJ's output `.class` files, a method invocation is used to call an aspect's advice

we can define pre- and post-conditions for advice for these states. In Figure 5 circles denote code blocks, black for advice and white for methods of P. Pre- and post-conditions are used to specify the expected and guaranteed states that a `before` or `after` advice can start or finish. α_b^{pre} and α_b^{post} refer to the pre- and post-condition of `before` advice on aspect α. Similarly α_a^{pre} and α_a^{post} for `after` advice.

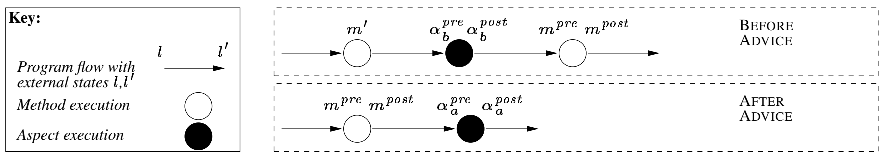

Fig. 5. Externally visible states for `before` and `after` advice

4.2 Executing User Defined Aspects in `conaj`

In order to understand the relationships and dependencies between advice contracts and method contracts, it is important to first expose the sequence of events that occur in a contracted OO program that also has user defined aspects. Given a `conaj` program with contracts and user defined aspects, what is the order of execution? Do you execute the user's aspect first, or the aspects generated by `conaj`? More importantly, which order of execution is *correct*?

To maintain the meaning of the method's pre- and post-conditions, the following conditions should be checked at specific points:

- Pre-conditions
  - The receiver's static type pre-condition should be the *first* one to be checked. In this way if the client failed to establish the correct state before calling m, this should be caught and reported back to the client.
  - The receiver's dynamic type pre-condition should be the *last* condition to be checked before executing m. In this way we guarantee that the pre-condition will be checked at the appropriate time before executing the first statement of m.
- Post-conditions
  - the *first* post-condition to be checked should be the one defined by the receiver's dynamic type. In this way we are actually checking that the method's implementation is meeting its obligations
  - the *last* post-condition to be checked should be the one defined by the receiver's static type. Since the caller is aware, and expects, the post-condition for the type of object that he initially called m on, to hold.

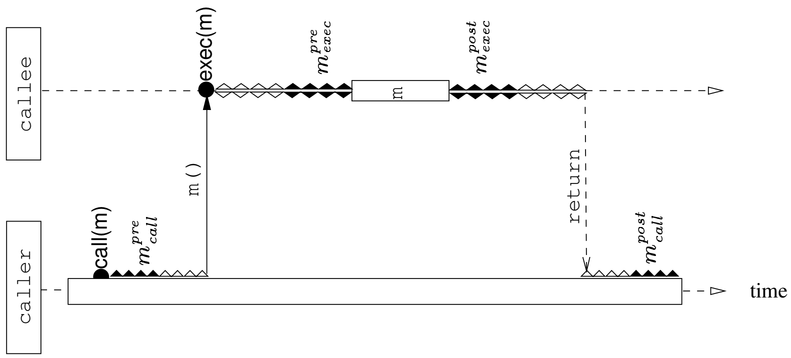

Fig. 6. m_{call}^{pre} denotes the pre-condition check for m's static type.m_{exec}^{pre} denotes the pre-condition check for m's dynamic type, and similarly for m_{call}^{post} and m_{exec}^{post}. call(m) and exec(m) denote joinpoints for call and execution respectively.

In Figure 6, the occurrence of an AspectJ joinpoint (call(m) and exec(m)) is shown as circles. The empty zig-zag lines denote available points for user aspects to be attached. The *filled* zig-zag lines are reserved points for `conaj` generated aspects that check pre- and post-conditions for method m. User aspects *cannot* advice the program at the points denoted with a filled zig-zag line. User aspects are "sandwiched" between `conaj` generated aspect. In this way a method's pre- and post-conditions will be checked at the intended execution points.

4.3 Pre- and Post-conditions for Before and After Advice

In this section, we propose an extension to the the syntax of the AspectJ language to accommodate pre- and post-conditions for `before` and `after` advice. One extra constraint on AOSD programs is that a single piece of advice is attached to each joinpoint. An extension of `conaj` for adding contracts to advice now takes as input aspect definitions with pre- and post-conditions. Through a preprocessing stage, aspects are generated to verify contracts on object instances *and* auxiliary class definitions to handle pre- and post-conditions on user defined advice.

First we present a sample program definition of advice with pre- and post-conditions along with a part of the generated output. An analysis of the logical implications that need to hold between pre- and post-conditions in advice and methods of the base system is then presented. An implementation is presented, using a set of judgments similar to those of Findler and Felleisen, as an extension to `conaj` enforcing runtime contract checking for pre- and post-conditions in advice.

Listing 1.9. An example of using pre- and post-conditions in the Logger aspect

```
aspect ContractLogger {
 pointcut logAacc(): call(* A.getSize(..));
 pointcut logAmut(): call(* A.setSize(..));

 before():
```

```
    logAacc(){
      @pre{target.getClass() == A.class}
      @post{true}
        System.out.println(" ++++ Loggin from the aspect GET ");
    }
  after():
    logAmut(){
      @pre{target.getClass() == A.class}
      @post{true}
        System.out.println(" ++++ Loggin from the aspect SET ");
    }
}
```

Listing 1.10. Sample output for `Logger` aspect showing the extra generated class definition and the wrapping of advice to check pre- and post-conditions

```
1  aspect ContractLogger {
2   pointcut logAacc(): call(* A.getSize(..));
3   pointcut logAmut(): call(* A.setSize(..));
4
5   before():
6     logAacc(){
7       if(thisJoinPoint.getTarget().getClass() == A.class){
8         (new ContractLogger_Before_PreCond()).logAcc(
               thisJoinPoint,this,thisJoinPoint.getTarget()
               );
9        System.out.println(" ++++ Loggin from the aspect
               GET ");
10       if (true){
11         (new ContractLogger_Before_PostCond).logAcc(this,
                thisJoinPoint.getTarget());
12       }else{
13        new Error(this,"before",jp);
14       }
15     }else{
16       new CompositionError(jp.getTarget().getClass().
               getName(),this,thisJoinPoint);
17     }
18   }
19 }
```

```
1  import org.aspectj.lang.*;
2
3  class ContractLogger_Before_PreCond {
4
5   public boolean logAcc(JoinPoint jp, ContractLogger
           uAspect, Object target){
6     String targetType = target.getClass().getName();
7     // Holds references to contract (aspects)
8     AspectMethInvoker aInv = AspectMethInvoker.getInst();
9
10    // aInv calls the appropriate contract methods using
           reflection
11    boolean m_pre = aInv.invoke(targetType, jp.
           getSignature(), jp.getArgs());
12    boolean res = target.getClass() == A.class;
13    if (!m_pre || res){
14     return res;
15    }
16    else {
17     new CompositionError(targetType,uAspect,jp);
18     return false;
19    }
20  }
21 }
```

Listing 1.10 shows an example usage of contracts inside advice. Listing 1.10 gives the resulting aspect and auxiliary class definition, dealing with contract enforcement for advice.

Pre- and post-conditions inside advice must be side effect free Java boolean statement. These expressions can refer to values that relate to the state of the aspect, and also to the state of receivers and callers of method invocations. All information concerning a joinpoint (e.g. `target`, `args`, `source`, etc.) can be refered from inside pre- and post-conditions.

The programmer, specifies the acceptable states in which advice may start/finish in. Failing to meet the pre-condition of a block of advice implies that the attachment of the specific aspect to the base program P is not correct (Listing 1.10 line 16). Similarly, if the post-condition of a piece of advice fails, this implies that the code inside the advice did not meet up to its obligations (Listing 1.10 line 13).

Once pre- and post-conditions have been satisfied, the way by which pre- and post-conditions of advice interplay with method calls (or executions) that they advice should be checked for its correctness. For before advice, upon completion of their advice code, control is passed to the original method m in P. Therefore, the post-condition of a before advice of an aspect α *must* imply the underlying method's (m) pre-condition (i.e., $\alpha_b^{post} \rightarrow m^{pre}$). On failure to do so, the error should lie with the aspect developer. The aspect's obligations conflict with the advised method's contract raising a composition error (Listing 1.10 line 17). This is an error in the logic of the advice either due to the aspect developer's misunderstanding of the advised method's contract, or due to an error in the pointcut definition of the aspect which allowed this attachment.

Similarly, a method's post condition *must* imply the attached aspects after advice pre-condition (i.e., $m^{post} \rightarrow \alpha_a^{pre}$). Failure to do so signals a composition error. The blame lies with the aspect programmer for failing to address a state of the program that was known in advance to be provided by the method after its completion. Finally, an `after` advice's post-condition must imply the underlying methods post-condition (i.e., $\alpha_a^{post} \rightarrow m^{post}$). Having this check will ensure that clients which use method m will be guaranteed the post-condition of the method. Clients expect m's post-condition to hold upon m's completion regardless of the presence of aspects in the system. Relaxing this implication will allow for aspects to break m's obligations causing unexpected behavior to clients of m. In this case the blame should lie with the aspect programmer for a malformed composition of his aspect.

4.4 Compiling Pre- and Post-conditions for `before` and `after` Advice

In this section, a more detailed definition of the compilation of contract checking rules for advice is provided. The rules build on top of `conaj`. Pre- and post-conditions of methods are being handled by `conaj`. Pre- and post-conditions for before and after advice are handled by newly generated classes. Wrapper methods are generated inside the user's aspect definition that perform the necessary calling conventions to check for pre- and post-conditions expressions and their implications to the underlying methods contract obligations.

As a final step, to enforce the appropriate execution of `conaj` generated aspects and user defined aspects, `dominates` statements are introduced to both `conaj` generated aspects and user defined aspects in order to maintain the correct execution sequence (Figure 6).

The rules provided are in the same flavor as [4]. Inside the rule definitions e_b $\text{PRE}_{\mathcal{A}}$ $\langle \alpha, \mathit{advice} \rangle$ is a relation between the set of user defined advice $\mathcal{A}$ and advice pre-conditions. The expression e_b is the pre-condition for *before* advice in aspect α. Similarly for e_a $\text{POST}_{\mathcal{A}}$ $\langle \alpha, \mathit{after} \rangle$ The BEFORE and AFTER rules in Figure 7 take user defined aspects containing pre- and post-conditions in `before` and `after` advice. Auxiliary class definitions are created with the bodies of before and after advice wrapped around conditionals Figure 8 (WRAP_{bef} and $\text{WRAP}_{\text{after}}$ respectively).

Finally Figure 9 shows the generation of all implications that are to be checked for advice pre- and post-conditions, and the underlying methods pre- and post-conditions. The judgments $jp \vdash c$ and $c \vdash m$ as well as the value of m_{pre} is being handled by `AspectMethInvoker` in the example output of Listing 1.10. The information is collected at runtime through the deployment of reflection both inside AspectJ's and Java's API.

Table 2. Rule definitions for contracts on advice

$P \rightharpoonup_p P'$
program P consisting of class and aspect definitions is compiled to
program P' with method contracts as aspect and extra classes for
contracts found in advice of user defined aspects.
made up of classes and methods

$P \vdash \mathit{defn} \rightharpoonup_d \mathit{defn}' \mathit{defn}_{pre}\ \mathit{defn}_{post}$
defn compiles into defn' with defn_{pre} and defn_{post} as as contract checkers

$\alpha \vdash \mathbf{before} \rightharpoonup_b \mathbf{before}'$
$\mathbf{before}$ gets compiled to $\mathbf{before}'$ which checks pre- and post-conditions defined in $\mathbf{before}$.
Contract violations blame α

$\alpha \vdash \mathbf{after} \rightharpoonup_a \mathbf{after}'$
$\mathbf{after}$ gets compiled to $\mathbf{after}'$ which checks pre- and post-conditions defined in $\mathbf{after}$.
Contract violations blame α

$P\ \alpha, c \vdash \mathit{advice} \rightharpoonup_{pre} \mathit{advice}'$
advice gets compiled to advice' which checks the pre-conditions defined in advice but also
the implication between the advice pre-condition and method's m pre-condition
defined in type c. Contract violations will blame α.

$P\ \alpha, c \vdash \mathit{advice} \rightharpoonup_{post} \mathit{advice}'$
advice gets compiled to advice' which checks the pre-conditions defined in advice but also
the implication between the advice pre-condition and method's m pre-condition
defined in type c. Contract violations will blame α.

$P\ \alpha, c \vdash e \rightharpoonup_e e'$
e gets compiled to e' which blames the composition of α with c, for contract violations

$jp \vdash c$
at the joinpoint jp an instance of class c is being adviced

$c \vdash m$
the method m is defined in class c

5 Discussion, Related Work and Future Work

Klaeren et al. [10] present *Aspect Composition Validation Tool* is presented that checks pre- and post-conditions for aspect compositions according to configurations of the system's components. Assertions in the form of pre- and post-conditions are used to validate aspect configurations in a system. The tool is developed using an older version of AspectJ (0.4beta7) which is drastically different than version 1.0.6. Aspects in version 0.4beta7 had the ability of being instantiated by the programmer, making method calls to aspects an easy task. Further more, a set of rules is added through AspectJ's introductions and composition is validated according to those rules. The correctness is defined to be a valid aspect configuration that will allow a receiver to perform its task as specified by the overall system specification. Unfortunately, checking that the behavior of attached aspects and the base system is well defined is not verified. As long as the composition of aspects is within the set of valid compositions, the system is correct. An aspect can therefore break a methods pre- or post-condition as long as the configuration at hand allows it. Therefore there is no clear distinction between a type's obligations

$$\textsc{Before}: \frac{\alpha \vdash \textbf{before}_j \rightharpoonup_b \textbf{before}'_j \qquad P,\alpha,c \vdash \textbf{before}_j \rightharpoonup_{pre} \textbf{before}'_{j_{\text{pre}}} \qquad P,\alpha,c \vdash \textbf{before}_j \rightharpoonup_{post} \textbf{before}'_{j_{\text{post}}} \quad \text{for } j \in [1,m]}{\begin{array}{l} P \vdash \textbf{aspect } \alpha \textbf{ extends } \alpha' \textbf{ implements } i_1 \ldots i_n\ pcd_1 \ldots pcd_m\ \textbf{before}_1 \ldots\ \textbf{before}_m \rightharpoonup_d \\ \quad \textbf{aspect } \alpha \textbf{ extends } \alpha' \textbf{ implements } i_1 \ldots i_n\ pcd_1 \ldots pcd_m\ \textbf{before}'_1 \ldots\ \textbf{before}'_m \\ \quad \textbf{class } check\_\alpha\_before\_pre\ \textbf{before}_{1_{pre}} \ldots\ \textbf{before}_{m_{pre}} \\ \quad \textbf{class } check\_\alpha\_before\_post\ \textbf{before}_{1_{post}} \ldots\ \textbf{before}_{m_{post}} \end{array}}$$

$$\textsc{After}: \frac{\alpha \vdash \textbf{after}_j \rightharpoonup_a \textbf{after}'_j \qquad P,\alpha,c \vdash \textbf{after}_j \rightharpoonup_{pre} \textbf{after}'_{j_{\text{pre}}} \qquad P,\alpha,c \vdash \textbf{after}_j \rightharpoonup_{post} \textbf{after}'_{j_{\text{post}}} \quad \text{for } j \in [1,m]}{\begin{array}{l} P \vdash \textbf{aspect } \alpha \textbf{ extends } \alpha' \textbf{ implements } i_1 \ldots i_n\ pcd_1 \ldots pcd_m\ \textbf{after}_1 \ldots\ \textbf{after}_m \rightharpoonup_d \\ \quad \textbf{aspect } \alpha \textbf{ extends } \alpha' \textbf{ implements } i_1 \ldots i_n\ pcd_1 \ldots pcd_m\ \textbf{after}'_1 \ldots\ \textbf{after}'_m \\ \quad \textbf{class } check\_\alpha\_after\_pre\ \textbf{after}_{1_{pre}} \ldots\ \textbf{after}_{m_{pre}} \\ \quad \textbf{class } check\_\alpha\_after\_post\ \textbf{after}_{1_{post}} \ldots\ \textbf{after}_{m_{post}} \end{array}}$$

Fig. 7. Class generation and wrapper methods for the elaboration of pre- and post-conditions in advice

through out the system, since the same call to an instance of the same type can behave differently depending on its aspect configuration. This feature obscures the understanding of a type's behavior inside the program, decreasing understandability.

Pipa [28] defines a Behavioral Interface Specification Language (BISL) tailored for AspectJ. Pipa statements extend the Java Modeling Language (JML) to accommodate pre- and post-conditions and invariants for advice. Specifications in Pipa, along with aspect definitions, are translated to JML and Java code, respectively. Using a BISL requires familiarity with yet another language with a full blown specification model increasing the complexity of its usage. In contrast, `conaj` is a straightforward extension of Java.

5.1 Future Work

Incorporating pre- and post-conditions for `around` advice comes as a necessary next step, along with a proof of soundness of our contracting rules for advice.

Implementing `conaj` in AspectJ 1.1 which now allows the advice *of* advice is another future step which will provide a natural extension to the design of `conaj` while at the same time providing a solution to a complex and interesting problem entirely within the domain of AOSD.

An interesting path for exploration, is the relationship between aspect interactions and contract enforcement and how can contracts specify ordering of aspect advice on joinpoints, making their interaction explicit but also providing a runtime check about a composition's correctness.

Another extension to the system, would be to allow the addition of user defined aspects that can be placed *before* or *after* contract checking operations for the base system's methods. This approach will allow for more interesting (or aggressive) AOSD extensions to a base program's behavior.

$$
\textsc{Wrap}_{\mathsf{bef}}:\frac{\begin{array}{cc} e_b\ \textsc{Pre}_{\mathcal{A}}\ \langle \alpha, \mathit{before}\rangle & e_a\ \textsc{Post}_{\mathcal{A}}\ \langle \alpha, \mathit{before}\rangle \\ P, \alpha, c \vdash e_b \rightharpoonup_e e'_b & P, \alpha, c \vdash e_a \rightharpoonup_e e'_a \end{array}}{\begin{array}{l} P, c, \alpha \vdash \textbf{before}\ (t_1\ x_1, \ldots, t_n\ x_n) : \mathit{pcd\_name}(x_1, \ldots, x_n)\ \mathit{stmts} \rightharpoonup_b \\ \quad \textbf{before}\ (t_1\ x_1, \ldots, t_n\ x_n) : \mathit{pcd\_name}(x_1, \ldots, x_n)\{ \\ \quad\quad \textbf{if}(e'_b)\{ \\ \quad\quad\quad (\textbf{new}\ \mathit{check\_\alpha\_before\_pre}).\mathit{pcd\_name}() \\ \quad\quad\quad \mathit{stmts} \\ \quad\quad\quad \textbf{if}\ (e'_a) \\ \quad\quad\quad\quad (\textbf{new}\ \mathit{check\_\alpha\_before\_post}).\mathit{pcd\_name}() \\ \quad\quad\quad \textbf{else Error}(\alpha, \mathsf{before}, \mathsf{post}); \\ \quad\quad \} \\ \quad\quad \textbf{else ComposeError}(\alpha, \mathsf{before}, \mathsf{pre}); \end{array}}
$$

$$
\textsc{Wrap}_{\mathsf{after}}:\frac{\begin{array}{cc} e_b\ \textsc{Pre}_{\mathcal{A}}\ \langle \alpha, \mathit{after}\rangle & e_a\ \textsc{Post}_{\mathcal{A}}\ \langle \alpha, \mathit{after}\rangle \\ P, \alpha, c \vdash e_b \rightharpoonup_e e'_b & P, \alpha, c \vdash e_a \rightharpoonup_e e'_a \end{array}}{\begin{array}{l} P, c, \alpha \vdash \textbf{after}\ (t_1\ x_1, \ldots, t_n\ x_n)\ \textbf{returning}\ \mathit{res} : \mathit{pcd\_name}(x_1, \ldots, x_n)\ \mathit{stmts} \rightharpoonup_a \\ \quad \textbf{after}\ (t_1\ x_1, \ldots, t_n\ x_n) : \mathit{pcd\_name}(x_1, \ldots, x_n)\{ \\ \quad\quad \textbf{if}(e'_b)\{ \\ \quad\quad\quad (\textbf{new}\ \mathit{check\_\alpha\_after\_pre}).\mathit{pcd\_name}() \\ \quad\quad\quad \textbf{let}\ \{\mathit{res} = \mathit{stmts}\} \\ \quad\quad\quad \textbf{in}\ \{ \\ \quad\quad\quad\quad \textbf{if}\ (e'_a) \\ \quad\quad\quad\quad\quad (\textbf{new}\ \mathit{check\_\alpha\_after\_post}).\mathit{pcd\_name}() \\ \quad\quad\quad\quad \textbf{else Error}(\alpha, \mathsf{after}, \mathsf{post}); \\ \quad\quad\quad\quad \mathit{res}\} \\ \quad\quad \}\textbf{else ComposeError}(\alpha, \mathsf{after}, \mathsf{pre}); \end{array}}
$$

Fig. 8. Wrapper method generation

6 Conclusion

This paper presents `conaj`, a preprocessor tool that implements Design by Contract using AspectJ. The paper also presents, as an extension to `conaj`, runtime contract checking for pre- and post-conditions for advice. Based on the ideas from Design by Contract, the paper analyzes and provides pre- and post-conditions for `before` and `after` advice. These conditions on advice code specify not only the expected states in which advice can start and finish, but also the relationship of these conditions to pre- and post-conditions defined in the underlying methods they advice.

The paper describes an extension to `conaj` that would allow for the runtime checks of pre- and post-conditions on advice. The enforcement of these conditions allow the explicit definition of advice behavior within the underlying system. One of the primary usage of advice is to extend an existing system. Any extension on a system through the incorporation of aspects *should* take into account the original system behavior and its obligations. The runtime contract checks of advice behavior against the base program's contracts, verifies that the behavior extensions provided by aspects *do not* change the base program's behavior to the extend where previously correct code is now rendered incorrect. Further more, error detection in AOSD programs can be extended to take into account behavior mismatch between advice and the base program.

Pre- and post-conditions in advice and their runtime check, allows for the incorporation of aspects without breaking already working code, while at the same time providing for better error reporting. Incorporation of pre- and post-conditions on `before` and `after` advice is a step forward towards reasoning about aspects and their behavior. The

$$\rightharpoonup_{pre}: \frac{jp \vdash c \qquad c \vdash m}{P, \alpha, c \vdash \textbf{before}(t_1\ x_1, \ldots, t_n\ x_n) : pcd\_name(x_1, \ldots, x_n)\{@\textbf{pre}\{e_b\}\ @\textbf{post}\{e_a\}\ stmts\} \rightharpoonup_{pre}}$$

$$\begin{array}{l}
\textbf{boolean}\ pcd_name(t_1\ x_1, \ldots, t_n\ x_n, \alpha\ this, c\ trg)\{ \\
\quad \textbf{let}\ \{m_{pre} = c_Contract.aspectOf().PreCond_m(trg, x_1, \ldots, x_n) \\
\qquad res = e_b\} \\
\quad \textbf{in if}(!m_{pre} || res) \\
\qquad res \\
\quad \textbf{else ComposeError}(\alpha, \mathsf{before}, \mathsf{pre})
\end{array}$$

$$\rightharpoonup_{pre}: \frac{jp \vdash c \qquad c \vdash m}{P, \alpha, c \vdash \textbf{after}(t_1\ x_1, \ldots, t_n\ x_n)\textbf{returning}\ res : pcd\_name(x_1, \ldots, x_n)\{@\textbf{pre}\{e_b\}\ @\textbf{post}\{e_a\}\ stmts\} \rightharpoonup_{pre}}$$

$$\begin{array}{l}
\textbf{boolean}\ pcd_name(t_1\ x_1, \ldots, t_n\ x_n, \alpha\ this, c\ trg, t\ res)\{ \\
\quad \textbf{let}\ \{m_{post} = c_Contract.aspectOf().PostCond_m(trg, res, x_1, \ldots, x_n) \\
\qquad ans = e_b\} \\
\quad \textbf{in if}(!m_{post} || ans) \\
\qquad ans \\
\quad \textbf{else ComposeError}(\alpha, \mathsf{after}, \mathsf{pre})
\end{array}$$

$$\rightharpoonup_{post}: \frac{jp \vdash c \qquad c \vdash m}{P, \alpha, c \vdash \textbf{before}(t_1\ x_1, \ldots, t_n\ x_n) : pcd\_name(x_1, \ldots, x_n)\{@\textbf{pre}\{e_b\}\ @\textbf{post}\{e_a\}\ stmts\} \rightharpoonup_{post}}$$

$$\begin{array}{l}
\textbf{boolean}\ pcd_name(t_1\ x_1, \ldots, t_n\ x_n, \alpha\ this, c\ trg)\{ \\
\quad \textbf{let}\ \{m_{pre} = c_Contract.aspectOf().PreCond_m(trg, x_1, \ldots, x_n) \\
\qquad res = e_a\} \\
\quad \textbf{in if}(!res || m_{pre}) \\
\qquad res \\
\quad \textbf{else ComposeError}(\alpha, \mathsf{before}, \mathsf{post})
\end{array}$$

$$\rightharpoonup_{post}: \frac{jp \vdash c \qquad c \vdash m}{P, \alpha, c \vdash \textbf{after}(t_1\ x_1, \ldots, t_n\ x_n)\textbf{returning}\ res : pcd\_name(x_1, \ldots, x_n)\{@\textbf{pre}\{e_b\}\ @\textbf{post}\{e_a\}\ stmts\} \rightharpoonup_{post}}$$

$$\begin{array}{l}
\textbf{boolean}\ pcd_name(t_1\ x_1, \ldots, t_n\ x_n, \alpha\ this, c\ trg, t\ res)\{ \\
\quad \textbf{let}\ \{m_{post} = c_Contract.aspectOf().PostCond_m(trg, res, x_1, \ldots, x_n) \\
\qquad ans = e_a\} \\
\quad \textbf{in if}(!ans || m_{post}) \\
\qquad ans \\
\quad \textbf{else ComposeError}(\alpha, \mathsf{after}, \mathsf{post})
\end{array}$$

Fig. 9. Pre- and post-condition compilation for advice

current implementation of `conaj` is available from *http://www.ccs.neu.edu/home/skotthe/conaj*.

References

1. D. Bartetzko, C. Fischer, M. Moller, and H. Wehrheim. Jass - java with assertions. In *Workshop on Runtime Verification, 2001. held in conjunction with the 13th Conference on Computer Aided Verification, CAV'01.*, 2001.
2. C. A. Constantinides and T. Skotiniotis. Reasoning about a classification of crosscutting concerns in object-oriented systems. In *Second International Workshop on Aspect-Oriented Software Development of the SIG Object-Oriented Software Development*, Bonn, February 21-22 2002. German Informatics Society.
3. E. Ernst and D. H. Lorenz. Aspects and polymorphism in AspectJ. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development*, pages 150–157, Boston, Massachusetts, Mar. 17-21 2003. AOSD 2003, ACM Press.

4. R. B. Findler and M. Felleisen. Contract soundness for object-oriented languages. In *Conferece on Object Oriented Programming, Systems, Languages and Applications*, number 11 in ACM SIGPLAN Notices, pages 1–15, Tampa Bay, Florida, October 2002.
5. R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *International Conference on Functional Programming (ICFP'02)*, pages 48–59, October 2002.
6. R. B. Findler, M. Latendresse, and M. Felleisen. Behavioral contracts and behavioral subtyping. In *Proceedings of ACM Conference Foundations of Software Engineering*, 2001.
7. R. W. Floyd. Assigning meaning to programs. In *Proceedings of Symposium on Applied Mathematics*, volume 19, Providence, R.I, 1976.
8. C. Hoare. An axiomatic basis for computer programming. In *Communications of the ACM*, 1969.
9. M. Karaorman, U. Hölzle, and J. Bruno. jContractor: A reflective Java library to support design by contract. *Lecture Notes in Computer Science*, 1616, 1999.
10. H. Klaeren, E. Pulvermüller, A. Rashid, and A. Speck. Aspect composition applying the design by contract principle. In *Proceedings of the GCSE 2000, Second International Symposium on Generative and Component-Based Software Engineering, 2000*, Oct 2000.
11. M. Kölling and J. Rosenberg. *Blue: Language Specification*, 1997.
12. R. Kramer. iContract - the java - design by contract - tool. In *Technology of Object-Oriented Langauges and Systems (TOOLS 98)*, August 1998.
13. G. T. Leavens, K. R. M. Leino, E. Poll, C. Ruby, and B. Jacobs. JML: notations and tools supporting detailed design in Java. In *Object-Oriented Programming, Systems, Languages, and Applications Companion*, pages 105–106, 2000. Also Department of Computer Science, Iowa State University, TR 00-15, August 2000.
14. K. J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, Boston, 1996.
15. B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.
16. C. Lopes, M. Lippert, and E. Hilsdale. Design by contract with aspect-oriented programming, 2002. U.S. Patent No. 06,442,750. Issued August 27,2002.
17. D. C. Luckham and F. von Henke. An overview of anna, a specification language for Ada. In *IEEE Software*, volume 2, pages 9–23, March 1985.
18. B. Meyer. *Object-Oriented Software Construction*. International Series in Computer Science. Prentice Hall, 1988.
19. B. Meyer. *Eiffel: The language*. Prentice Hall PTR, 1991.
20. S. M. Omohundro. The Sather 1.0 specification. Technical Report TR-94-062, International Computer Science Institute, Berkeley, 1994.
21. P.America. Designing an object-oriented languages with behavioural subtyping. In J. de Bakker, editor, *Foundations of Object Oriented Languages*, volume 489 of *Lecture Notes in Computer Science*, pages 60–90. Springer, 1991.
22. D. Rosenblum. A practical approach to programming with assertions. In *IEEE Transactions on Software Engineering*, volume 21(1), pages 19–31, January 1995.
23. T. Skotiniotis, K. Lieberherr, and D. H. Lorenz. Aspect instances and their interactions. In *AOSD 2003 Workshop on Software-engineering Properties of Languages for Aspect Technologies*, Boston, Massachusetts, Mar.18 2003. AOSD 2003 Workshop on Software-engineering Properties of Languages for Aspect Technologies, ACM Press.
24. The AspectJ Team. AspectJ Development Tools, 2003. http://www.eclipse.org/aspectj/.
25. The AspectJ Team. *The AspectJ Programmers Guide*, 2003. http://www.eclipse.org/aspectj/.
26. The AspectJ Team. The AspectJ Web Site, 2003. http://www.eclipse.org/aspectj/.
27. The Demeter Group. *DemeterJ Resources*, 2003.

28. J. Zhao and M. Rinard. Pipa: Behavioral Interface Specification Language for AspectJ. In *Proceedings of Fundamental Approaches to Software Engineering (FASE'2003)*, pages 150–165, 2003.