# Conaj:Generating Contracts as Aspects

Therapon Skotiniotis and David H. Lorenz*

College of Computer and Information Science,
Northeastern University,
360 Huntington Avenue 161 CN,
Boston, Massachusetts 02115 USA.
{skotthe,lorenz}@ccs.neu.edu

**Abstract.** Design by Contract (DBC) is a well established practice in Object-Oriented (OO) programming. The need to intervene at well defined points in the execution of a program in order to verify contract definitions make the enforcement of DBC a crosscutting concern. The recent results of Findler and Felleisen [1] have shown that DBC in OO languages is often been erroneously enforced. CONAJ is a tool developed using generative programming (GE) and aspect-oriented software development (AOSD) techniques, providing support for DBC in the Java programming language following the semantics of Findler and Felleisen. Aspects are generated to capture contract definitions and their enforcement. In this paper we show that it is possible with current AOSD technologies to capture the DBC concern using aspects. The paper discusses issues in the design of generic aspects, technological limitations and their solutions. A templated design of how an AOSD solution to DBC can be achieved with AspectJis presented. Beyond the experience gained by the development of CONAJ using AOSD tools, the paper provides an AOSD implementation of a non-trivial crosscutting concern illustrating the shortcomings of current AOSD technologies. It provides not only an interesting benchmark application for AOSD, but also a verification of some of the claims made in favor of AOSD.

## 1 Introduction

Design by Contract (DBC), first introduced in Object-Oriented (OO) languages by the Eiffel community [2], provides a mechanism to define and check at runtime contract conditions (pre-conditions, post-conditions and invariants). Recent work [1] has exposed a relationship between contracts and the notion of behavioral subtyping in OO languages. Specifically, beyond simply checking a method's pre- and post-condition an extra check on the class hierarchy has to be enforced. The hierarchical check performed ensures at runtime that an overridden method's pre-condition is implied by the super-type's pre-condition (super.m$_{pre}$ $\rightarrow$ sub.m$_{pre}$). Similarly the reverse relationship must hold for post-conditions sub.m$_{post}$ $\rightarrow$ super.m$_{post}$. Findler et. al. also provide appropriate blame assignment extended to blame not only the caller (when a pre-condition fails) and the callee (when a post-condition fails) but also the implementor of a class (when

* Supported in part by the National Science Foundation (NSF) under Grant No. CCR-0204432, and by the Institute for Complex Scientific Software at Northeastern University.

a hierarchy check fails) for not providing a proper subtype implementation. With the exception of Jass [3], all other DBC tools including Eiffel [2, 4–7] do not incorporate hierarchical checks as described by [1].

Runtime contract checking needs to intervene at systematically well defined execution points, giving it a crosscutting flavor. Contracts can be mapped to aspects, pre-conditions to before advice and post-conditions to after advice, with the points of interventions being expressed as pointcuts, this observation has been made by others in the [8–10]. However, there is no published implementation of an AOSD-based DBC tool that verifies this claim.

Providing such a non-trivial AOSD solution, puts to the test expectations such as ease of extension, better modularity and understandability. Some important issues that an AOSD solution to DBC explores are:

- Interesting interactions amongst aspects required to implement the hierarchy runtime contract checks.
- The ability and expressiveness of pointcuts to capture situations where the runtime type of an instance is different from the its static type. In these situations hierarchy checking should start from the dynamic type and *not* the static type of the object.
- Obtain information about the caller and callee for error reporting from inside an aspect.

ConAJ was developed as an AOSD DBC tool for Java generating aspect definitions that enforce runtime contract checking as per [1]. The theoretical foundation is reported elsewhere [11]. Hoping to leverage the claim of ease of extensibility of an AOSD solution, our work in [11] deals with contracts for advice. In this paper, we concentrate on the generative programming issues faced during the development of ConAJ where the originally envisioned AOSD solution could not be materialized using one of the most popular aspect oriented languages (AspectJ) without generics.

Section 2 gives the design of ConAJ, its architecture, the skeleton of the generated aspects that capture runtime contract checks as well as an example in ConAJ. Section 3 gives a detailed analysis of the template used to generate aspects implementing runtime contract checks in AspectJ that can be used with other OO languages and aspect languages designed along the same design dimensions as AspectJ [12–16]. Section 4 provides a short discussion of lessons learned through the development of ConAJ. Section 5 talks about related work and section 6 concludes.

## 2 Aspects for Contracts

The key idea in using aspects as contracts is to provide a solution that encapsulates both the code that specifies contracts and the code implementing the contract checking mechanism for enforcing proper subtype behavior as defined by [1]. While several contract checking tools have explored alternatives other than AOP [4, 5, 3, 2], an AOSD implementation is better suited for DBC due to the crosscutting nature of runtime contract checking [8–10]. A requirement imposed on the aspectual design of ConAJ was that code which deals with contracts is solely handled by aspects. Given an OO program along with contract annotations for each type, the solution should only introduce new “components” as aspects and keep the original classes unaffected.

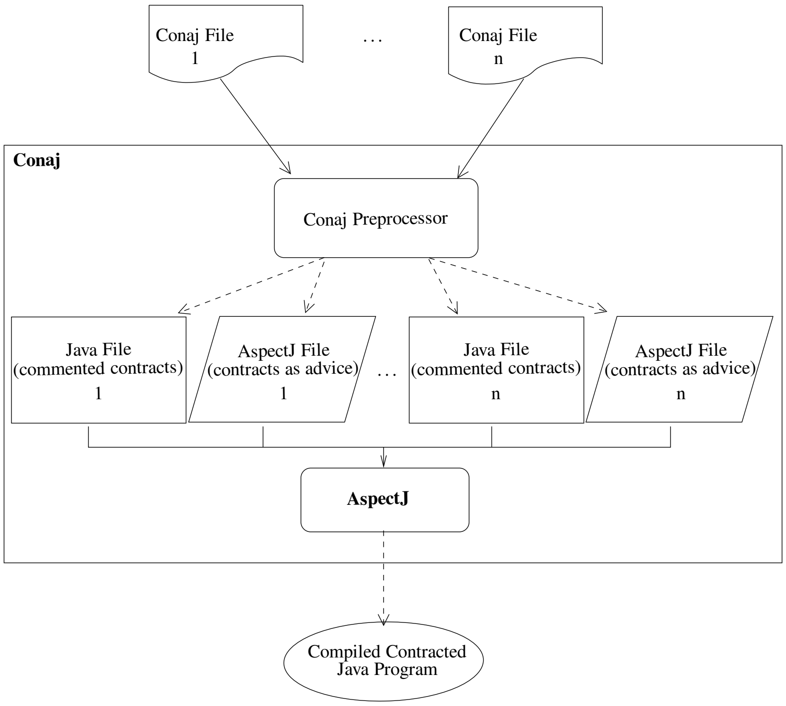


**Fig. 1.** CONAJ Architecture

### 2.1 Architecture

CONAJ (Figure 1) is a preprocessing tool, implemented using DemeterJ.[1] 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.

### 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`**,

[1] CONAJ was developed with DemeterJ version 0.8.6 [17, 18] and AspectJ [19] version 1.0.6

and **`old`**. The specification of CONAJ's grammar (Figure 2) uses DemeterJ's [18] syntax for class files, which can be viewed as a textual representation of a UML class diagram. Class names denoted by '[ ]' are optional. Rules with '=' denote concrete classes and their members and ':' denotes inheritance from abstract classes. Definitions of **`@pre`**

```
                /* Extensions for Conaj */
Pre           = @pre{ ConditionalOrExpr } .
Post          = @post{ ConditionalOrExpr } .
Invariant     = @invariant{ ConditionalOrExpr } .
ClassBody     = { [Invariant] ClassBody [Invariant] } .
InterfaceDecl = interface Identifier [ extends NList ]
                  { [Invariant] IDecls [ Invariant ] }.
MethodDecl    = MethodModifiers MethodSig AnyBlock .
AnyBlock      = A_Block | A_SemiColon .
A_Block       = { [ BlockHead ] BlockStatements
                  [ BlockTail ] } .
BlockHT       : BlockTail | BlockHead
                  common [ Pre ] [ Post ] .
BlockHead     = .
BlockTail     = .
A_SemiColon   = ; [ Pre ] [ Post ] .
                /* Java Language Grammar. (omitted) */
```

**Fig. 2.** CONAJ Grammar (using DemeterJ Class Graph syntax)

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 obtained (if the method returns a value other than `void`) after the method $m$ completes its execution. Listing 1.1 provides the implementation of a stack in `conaj`.

### 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. Generated aspects follow a logical structure consisting of four pieces

**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();}
}
```

1. The aspect heading defining the name of the aspect and a default clone operation for the type that this aspect will attach to.
2. A section where all execution points where runtime contract checking is to take place is expressed using pointcut definitions.
3. A section of methods, some of which hold the boolean expressions provided inside contract definitions. Extra methods capture the hierarchy runtime checking mechanism.
4. A section of before and after advice that will initiate the calls to the aspect generated methods implementing a method's pre-condition and post-conditions along with the necessary hierarchy checking. Before advice deals with pre-conditions and after advise deals with post-conditions.

More specifically, in the case of an invariant definition a 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 executions. Method call join points are used to check contracts before method dispatch, method execution join points are used to check contracts after method dispatch. 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-

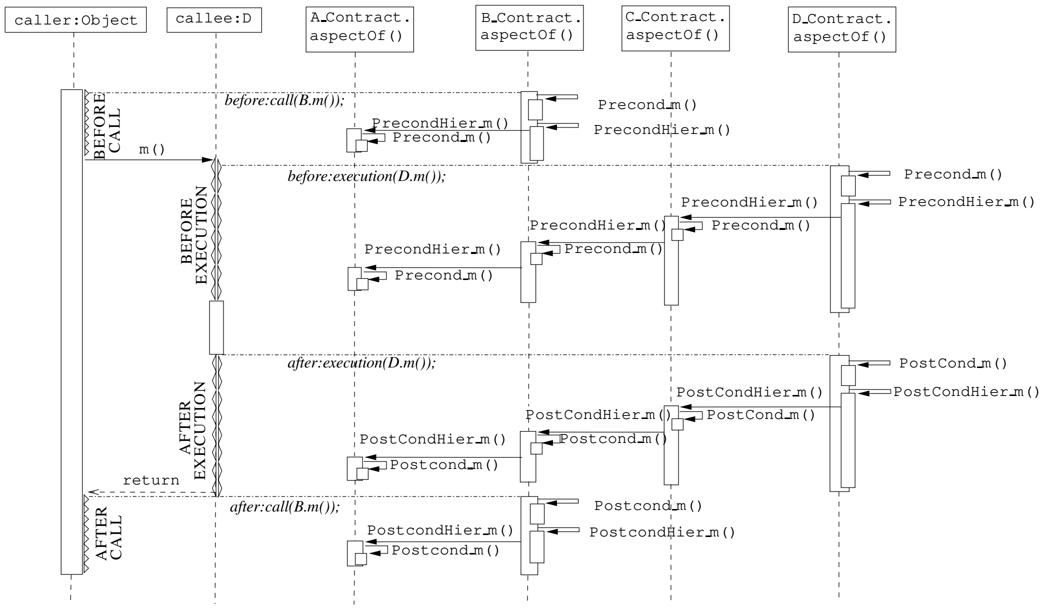


**Fig. 3.** Interaction diagram for the code fragment { `B b = new D(); b.m();` }

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. 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 symmetric case of post-conditions on $m$. The hierarchy checking for both the static and the dynamic type do check the same conditions twice. This repetition is not accidental but *required* for situations where components (e.g., user defined aspects) can intervene and alter the systems state. This checking is needed for the extensions to ConAJ for the provision of DBC in a system with user aspect definition discussed in a related paper [11].

A closer look at Figure 3 reveals a repetitive check of conditions between `A_Contract` and `B_contract`. This repetition will happen for any situation where the runtime type of an object differs from its static type. However the repetition is not accidental. ConAJ has been extended to deal with systems where user defined aspects are present. In the presence of user defined aspects [11], advice may be executed after ConAJ completes the pre-condition hierarchy checking of the static type and before ConAJ's pre-condition hierarchy checking of the object's dynamic type. This intervention can alter the caller, callee and/or the arguments of the call. Therefore these checks *must* be performed this way since other entities (e.g., aspects, reflective systems) may intervene causing a previously checked condition to be falsified. Listing 1.1 shows an example of a Stack implemented in ConAJ, and Listing A (in the appendix) the generated aspect. A detailed analysis of how ConAJ generates aspects to implement method contracts is given in the next section.

## 3 The Anatomy of an Aspect Implementing a Contract

This section provides the details of CONAJ's aspect generation. 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 of the form $\langle$TERM$\rangle$ denote type specific information. The meta variables (Table 1) are place holders that will be filled

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

| *Terms* | *Definition* |
|---|---|
| $\langle$Type$\rangle$ | Any Java or user defined type name. The special variable $\mathcal{T}$ refers to the type currently being processed (e.g `MyStack`) |
| $\langle$Method$\rangle$ | 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$\rangle$ | 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 |
| $\langle$Method$_i$ Pre-code$\rangle$ | The segment of Java code provided as the precondition to $\langle$Method$_i\rangle$ |
| $\langle$Method$_i$ Post-code$\rangle$ | The segment of Java code provided as the post-condition to $\langle$Method$_i\rangle$ |
| $\langle\mathcal{FA}\rangle$ | A list of *formal parameters* $\langle$Type$_1\rangle\langle$Arg$_1\rangle\ldots\langle$Type$_n\rangle\langle$Arg$_n\rangle$ |
| $\langle\mathcal{A}\rangle$ | A list of argument names (e.g $\langle$Arg$_1\rangle\ldots\langle$Arg$_n\rangle$) |
| $\langle$Arg$\rangle$ | A name binding for a method/pointcut argument |

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

**Listing 1.2.** Aspect definition and the introduction of `clone`

```
privileged aspect ⟨T⟩_Contract {

  declare parents : ⟨T⟩ implements Cloneable;
  public Object ⟨T⟩.clone(){
    //Implementation of "shallow" cloning for usage with old
  }
  //Reference previous state of object
  ⟨T⟩ old;

  pointcut scope(): !within(⟨T⟩_Contract)
    && !cflow(withincode(* ⟨T⟩_Contract.*(..)));
```

Listing 1.2 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 default 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 gives the two pointcuts that are defined for each method $\langle$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 $\langle$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 [20] 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 1.3 is instantiated for each method $Method_I$ in $\mathcal{T}$.

**Listing 1.3.** Pointcut definitions for each method

29 // for each $Method_i$ defined in $\mathcal{T}$ with either a **`@pre`** or **`@post`**
30 **pointcut** $\langle\mathcal{T}\rangle$_$\langle$Method$_i\rangle$($\langle\mathcal{T}\rangle$ tInst,$\langle\mathcal{FA}\rangle$):
31 **call**( **public** $*$ $\langle\mathcal{T}\rangle$.$\langle$Method$_i\rangle$(..))
32 && !**call**(**public** $*$ ($\langle\mathcal{T}\rangle$+ && !$\langle\mathcal{T}\rangle$).$\langle$Method$_i\rangle$(..))
33 && **args**($\langle\mathcal{A}\rangle$) && **target**(tInst) && scope();
34
35 **pointcut** dynamic_$\langle\mathcal{T}\rangle$_$\langle$Method$_i\rangle$($\langle\mathcal{T}\rangle$ tInst,$\langle\mathcal{FA}\rangle$):
36 **execution**(**public** $*$ $\langle\mathcal{T}\rangle$.$\langle$Method$_i\rangle$(..))
37 && !**execution**( **public** $*$ ($\langle\mathcal{T}\rangle$+ && !$\langle\mathcal{T}\rangle$).$\langle$Method$_i\rangle$(..))
38 && **args**($\langle\mathcal{A}\rangle$) && **target**(tInst) && scope();
39 // similar pointcut definitions for each of the remaining methods of $\mathcal{T}$ (ommitted)

Listing 1.4 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 1.4.** 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 1.2, 1.3 and 1.4 make up all the pointcuts that an aspect (implementing a contract definition) will contain.

Listing 1.5 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[2].

**Listing 1.5.** 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();
    res=Precond_⟨Method_i⟩(tInst,⟨A⟩);
    next=PrecondHier_⟨Method_i⟩(tInst,⟨A⟩) ;
    if (!res){
      // throw PreCond Error exception⟨Method_i⟩
    }
    if (!next) {
      // throw HierPreCond Error exception⟨Method_i⟩
    }
  }
before(⟨T⟩ tInst,⟨FA⟩):
  dynamic_⟨T⟩_⟨Method_i⟩(tInst,⟨A⟩){
    hierResult=PrecondHier_⟨Method_i⟩(tInst,⟨A⟩);
    res=Precond_⟨Method_i⟩(tInst,⟨A⟩);
    if (!(!hierResult || res)){
      // throw HierPreCond Error exception
    }
  }
```

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

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 92–103). 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 [1], 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 104–111). The code checks (using the same methods as those given above) for the method's precondition and that the hierarchy is well formed.

**Listing 1.6.** After Advice

```
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) {
    //throw HierPostCond Error exception
  }
}

after(⟨T⟩ tInst,⟨FA⟩) returning(⟨Type⟩ result):
  dynamic_⟨T⟩_⟨Method_i⟩(tInst,⟨A⟩){
    postResult=Postcond_⟨Method_i⟩(tInst,result,⟨A⟩);
    if(!postResult){
      //throw PostCond Error exception
    }
    postHier=PostcondHier_⟨Method_i⟩(tInst,postResult,result,⟨A⟩);
    if(!postHier){
      // throw HierPostCond Error exception
    }
  }
after(⟨T⟩ tInst):
  ⟨T⟩_invariant(tInst){
    if (!Invariant(tInst)){
      // throw Invariant Error exception
```

```
151       }
152     }
```

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 1.6 shows the three types of after advice that are generated.

Lines 147–152 checks the invariant condition for type $\mathcal{T}$ by calling the appropriate method inside the aspect. The first after advice in Listing 1.6 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 [1]. Inside the aspect this is done recursively by the calls to the `PostCondHier_`$\langle$Method$_i\rangle$`()`. The last type of after advice (lines 136–146) 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 1.7.** 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;}

//for each method with a postcondition definition
 public boolean Postcond_⟨Method_i⟩(⟨𝒯⟩ tInst,⟨Type⟩ result,⟨𝓕𝓐⟩){
  (⟨Method_i Postcondition code⟩)? return true : return false;}

//for each method with a precondition definition
 public boolean PrecondHier_⟨Method_i⟩(⟨𝒯⟩ tInst,⟨𝓕𝓐⟩){
  myPre = Precond_⟨Method_i⟩( tInst,⟨𝓐⟩);
  hierarchy=
  ⟨SuperType_1⟩_Contract.aspectOf().PrecondHier_⟨Method_i⟩(tInst,⟨𝓐⟩)
    ||. . .
    ||
    ⟨SuperType_k⟩_Contract.aspectOf().PrecondHier_⟨Method_i⟩(tInst,⟨𝓐⟩);
  (!hierarchy || myPre) ? return myPre : return false;
 }

//for each method with a postcondition definition
 public boolean PostcondHier_⟨Method_i⟩(⟨𝒯⟩ tInst,boolean last, ⟨Type⟩ result,⟨𝓕𝓐⟩){
  myPost=Postcond_⟨Method_i⟩(tInst,result,⟨𝓐⟩);
    if (!last || myPost) { // last => myPost
  return
```

```
237         (⟨SuperType1⟩_Contract.aspectOf().PostcondHier_⟨Methodi⟩(tInst,myPost,result,⟨A⟩))
238         && ...
239         &&
240         ⟨SuperTypek⟩_Contract.aspectOf().PostcondHier_⟨Methodi⟩(tInst,myPost,result,⟨A⟩);
241       }else
242         return false;
243   }
```

Listing 1.7 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 210–219). The hierarchy checking methods (lines 222 to 243) 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.

## 4 Lessons Learned

### 4.1 AspectJ Limitations

In developing CONAJ, certain decisions concerning the solution's design and 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 that performs as expected, but cannot be inferred by the documentation of AspectJ at the time. The idiom helped in capturing situations where the runtime type of an instance was different from its static type without any extra program analysis.

The design could have been more robust if the ability to dispatch on aspect method or advice names were possible [21]. That would have allowed an aspect hierarchy identical to the hierarchy of types to be created where only the more specific (according to target type) pointcut would have been in effect at program execution time. Hierarchy checking could then have been implemented in a similar manner as in current OO languages through the usage of `super` inside an aspect to refer to its super aspect's advice. Currently, CONAJ keeps track of aspect names and type names. It inserts the appropriate calls in order to obtain the correct aspect instances when traversing the hierarchy of types in order to check for the correct contract implications between overridden methods.

Aspect definitions cannot refer (call) advice code found in another aspect. This was also the reason of placing all pre-condition, post-condition and invariant code into methods inside an aspect. This was the only way code from one aspect can call code found in another aspect. The ability to provide code inside advice to other aspect entities would have allowed for aspect definitions to be more compact increasing the understandability of aspect definitions.

Still, AspectJ features have helped the development of CONAJ. The extra reflective power provided through AspectJ's library allows for 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 can be obtained via AspectJ's

`JoinPoint` interface). Information that assisted both in error reporting but also acquiring with greater ease information about the running program.

### 4.2 AOSD Shortcomings

The aim of AOSD is to provide for an advanced separation of concerns and address the limitations of existing technologies that lead to tangled implementations of developer concerns. Kienzel et. al. [22] the authors attempt to verify the claims made in favor of AOSD by attempting to design a distributed application where transactions and synchronization are captured as aspects in the system. The results show that separation of concerns can not be achieved to the desired level, the separation that can be achieved is simply syntactical making the overall system hard to reason about.

Our experiences with CONAJ partly overlap with [22]. Reasoning about the system as a whole was made more difficult in the presence of aspects. Development tools provided by the AspectJ team assist in understanding your system, but difficulties in understanding are not completely overcome. In more complex AOSD systems where pointcuts are not trivial, understanding were exactly in the program's execution an aspect is activated is difficult. Developers have to jump from code in aspects to code in classes (and vice versa), and these jumps do not always depend on static properties of the code. It is therefore harder to understand how the whole system behaves by inspecting its components.

In terms of separation of concerns, in the case of DBC and runtime contract enforcement the separation was achieved. Runtime contract checking was entirely done with aspects, and further more, the mechanism can be automated via generative programming.

## 5 Related Work

Enforcing runtime contract checking in OO languages has been provided in many OO languages using different implementations, as part of the implementation [2, 6, 7], or as an add-on library with a preprocessor [3–5].

Incorporating DBC as part of the language provides access to static information as well as to information gathered by other phases of the compilation process. Information of static types, caller and callee static types etc. On the down side, alterations to the mechanism itself require to alter part of the compiler and the data dependencies between the DBC mechanism and the rest of the compilations stages. Evolution of the system becomes a complex task.

Providing runtime contract checking mechanism via an add-on library allows one to extend a language with DBC capabilities event hough the language was not originally developed to accommodate DBC. The extension comes with a extension to the language's syntax and further analysis in order to obtain information concerning static types from source code. This duplicates work since this is already done by most language compilers [23]. Further more, the generation of contracts is captured in both new components (i.e. classes) but also the necessary calls to these new entities have to be inserted in the original class definitions. The "seeding" of these calls in the original class definitions generates scattered code, but also an implementation difficulty of inserting code at the appropriate source locations. Extensibility of add-on libraries still requires re-engineering part of the library as well as the syntax analysis (in the case the extension affects the input syntax) or even worse the locations where new code will be inserted. An exception to the set of tools using an add-on library approach is jContractor [5] where the system is implemented using a Java reflection. Contract definitions are restricted to a specific naming convention so as to allow for their easy retrieval via reflection. All calls to the components that implement the contract checking mechanism are done through reflection and thus the amount of "seeding" calls is minimized. The design however constraints the syntax of contract definitions making their usage brittle.

In CONAJ, due to the usage of DemeterJ and AspectJ some of the issues faced by other implementations of DBC were easily overcome. DemeterJ allows for an iterative approach to extending both the language syntax to accommodate contract definitions, but also the analysis of the parsed in code via the usage of class graphs and adaptive visitors. The usage of AspectJ made the identification of all points of execution easier through the usage of pointcut definitions. Distinction between static and runtime types of instances is achieved through two pointcut definitions. The difficulties of pinpointing where to add the extra checking code for contracts was easily achieved through advice, and the extra reflective facilities of AspectJ provided all necessary information required for blame assignment and error reporting. As a result CONAJ is easier to extend both in terms of syntax but also in terms of the mechanism itself.

## 6 Conclusion

In this paper we show generative programming and AOSD are used in concert to develop a DBC mechanism on top of Java. The recognition that DBC *is* a crosscutting concern has been identified by others in the community, however, a complete implementation based on AOSD was never attempted before. Recent results from Findler and Felleisen have shown how current DBC tools for OO languages where flawed and in their work provide the semantics for a correct runtime contract checking tool.

The implementation of such a non-trivial crosscutting concern is a challenge for AOSD and current AOSD technologies. CONAJ is a DBC tool developed using generative and AOSD technologies (DemeterJ and AspectJ ) and following the semantics proposed by [1]. Unlike [22] DBC is a suitable crosscutting concern that can be separated allowing for better modularity and ease of extension. The usage of DemeterJ and adaptive programming allow for an iterative approach to syntax extensions and analysis of contract definition for the generation of AspectJ code. AspectJ allows the capture of all cases where a DBC mechanism should intervene to verify pre-conditions, post-conditions and invariants. Further more, AspectJ provides extended reflective information (e.g., arguments values and types, caller and callee static and dynamic types etc.) that also assist in error reporting and in blame assignement.

There are however some limitations. The limitation of design choices for the generated aspects stemmed from limitations in AspectJ. The absence of aspect polymorphism [21] along with the inability to refer to advice outside of its aspect definition imposed certain design decisions. Still, these limitations can be resolved by means of generative programming.

We hope with this paper to also introduce the enforcement of DBC as a new non-trivial aspect oriented benchmark for the AOSD community. CONAJ is available from `http://www.ccs.neu.edu/home/skotthe/conaj/`.

## References


1. Findler, R.B., Felleisen, M.: Contract soundness for object-oriented languages. In: Conferece on Object Oriented Programming, Systems, Languages and Applications. Number 11 in ACM SIGPLAN Notices, Tampa Bay, Florida (2002) 1–15
2. Meyer, B.: Eiffel: The language. Prentice Hall PTR (1991)
3. Bartetzko, D., Fischer, C., Moller, M., Wehrheim, H.: Jass - java with assertions. In: Workshop on Runtime Verification, 2001. held in conjunction with the 13th Conference on Computer Aided Verification, CAV'01. (2001)
4. Kramer, R.: iContract - the java - design by contract - tool. In: Technology of Object-Oriented Langauges and Systems (TOOLS 98). (1998)
5. Karaorman, M., Hölzle, U., Bruno, J.: jContractor: A reflective Java library to support design by contract. Lecture Notes in Computer Science **1616** (1999)
6. Kölling, M., Rosenberg, J.: Blue: Language Specification. (1997)
7. Omohundro, S.M.: The Sather 1.0 specification. Technical Report TR-94-062, International Computer Science Institute, Berkeley (1994)
8. Lopes, C., Lippert, M., Hilsdale, E.: Design by contract with aspect-oriented programming (2002) U.S. Patent No. 06,442,750. Issued August 27,2002.
9. Constantinides, C.A., Skotiniotis, T.: 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, German Informatics Society (2002)
10. The AspectJ Team: The AspectJ Programmers Guide. (2003) `http://www.eclipse.org/aspectj/`.
11. Skotiniotis, T., Lorenz, D.H.: From contracts to aspects and back. Submitted to OOPSLA 2004 (2004)
12. Hirschfeld, R.: AspectS—aspect-oriented programming with Squeak. In Aksit, M., Mezini, M., Unland, R., eds.: Architectures, Services, and Applications for a Networked World. Number 2591 in Lecture Notes in Computer Science, Springer Verlag (2003)
13. de Alwis, B.: Apostle (2004) `http://www.cs.ubc.ca/labs/spl/projects/apostle/`.
14. Distributed Systems Group, T.C.D.: Aspectc sharp (2004) `http://www.dsg.cs.tcd.ie/index.php?category_id=169`.
15. Spinczyk, O., Gal, A., Schröder-Preikschat, W.: Aspectc++: an aspect-oriented extension to the c++ programming language. In: Proceedings of the Fortieth International Confernece on Tools Pacific, Australian Computer Society, Inc. (2002) 53–60
16. Rajan, H., Sullivan, K.: Eos: instance-level aspects for integrated system design. In: Proceedings of the 9th European software engineering conference held jointly with 10th ACM SIGSOFT international symposium on Foundations of software engineering, ACM Press (2003) 297–306
17. Lieberherr, K.J.: Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns. PWS Publishing Company, Boston (1996)
18. The Demeter Group: DemeterJ Resources. (2003)

19. The AspectJ Team: The AspectJ Web Site (2003) `http://www.eclipse.org/aspectj/`.
20. Skotiniotis, T., Lieberherr, K., Lorenz, D.H.: Aspect instances and their interactions. In: AOSD 2003 Workshop on Software-engineering Properties of Languages for Aspect Technologies, Boston, Massachusetts (2003)
21. Ernst, E., Lorenz, D.H.: Aspects and polymorphism in AspectJ. In: Proceedings of the $2^{nd}$ International Conference on Aspect-Oriented Software Development, Boston, Massachusetts, AOSD 2003, ACM Press (2003) 150–157
22. Kienzle, J., Guerraoui, R.: AOP: Does it make sense? the case of concurrency and failures. In: Proceedings ECOOP '2002. Volume 2374 of LNCS., Springer Verlag (2002)
23. Friedman, D.P., Wand, M., Haynes, C.T.: Essentials of Programming Languages. Second edn. MIT Press, Cambridge, MA (2001)

## A MyStack_Contract.aj

```
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;
```

```
    }
}
```