3 CoffeeStrainer Explained

This chapter explains how constraints can be expressed with CoffeeStrainer. In Section 3.1, the basic ideas of the CoffeeStrainer framework are introduced. Section 3.2 explains how static constraints can be implemented, and Section 3.3 explains CoffeeStrainer's support for dynamic constraints.

3.1 CoffeeStrainer basics

CoffeeStrainer's main contribution is to support static constraints by making the program's structure available to constraints that check for certain structural properties. This section introduces to the CoffeeStrainer framework by explaining how the program's structure is represented (Section 3.1.1), by walking through a simple example constraint (Section 3.1.2), and finally, by giving an overview of the different kinds of static constraints which can be implemented with CoffeeStrainer (Section 3.1.3).

3.1.1 The abstract semantics graph

In CoffeeStrainer, the static structure of a program is represented by the *abstract semantics graphs (ASGs)* of the program's compilation units. In Java, compilation units are single files which can be compiled independently by a compiler. The abstract semantics graph of a Java compilation unit is based on the *abstract syntax tree* built along the Java grammar. To build an abstract syntax tree, each occurrence of a terminal symbol is represented by a leaf node of the tree, and each occurrence of a nonterminal symbol is represented by an inner tree node; consequently, the root node of the abstract syntax tree represents the whole compilation unit. This tree, as explained below, is augmented with information obtained from *name analysis* and *type analysis*, turning the abstract syntax tree into an abstract semantics graph [Devanbu et al. 1996]. The ASG built by CoffeeStrainer completely represents one compilation unit.

By name analysis, each use of a name is associated with its definition. For example, the leaf node representing a local variable access is associated with the tree node that represents the local variable's declaration, and, similarly, the tree node representing the local variable's declaration is associated with the tree node representing the local variable's declared type. Note that this type may be defined in a different compilation unit, and thus that the association may be with a node of a different abstract semantics graph. However, because the distinction between different abstract semantics graphs is important only when considering whether or not a system supports separate compilation, the set of abstract semantics graphs

of a program often may be viewed as a single ASG which is formed by one (virtual) root node representing the program whose child nodes are the nodes representing the different compilation units of the program.

By type analysis, each tree node representing an expression is associated with its static type which is determined according to the type rules of the programming language. In Java, an expression's type is completely determined by the constituents of the expression and does not depend on the expression's context.

In Figure 3.1, an example program together with its ASG is shown. The tree part of the ASG is drawn using normal connection lines; the graph references obtained from name and type analysis are drawn as dashed arrows. The example program consists of two classes `A` and `B` in a package `p`. Both classes contain one field declaration; field `f` in class `A`, and field `g` in class `B`, which has an initializer expression. The tree part of the ASG consists of the nodes and the solid lines, while additional references obtained from name and type analysis are shown using dashed arrows. There are dotted arrows from both node `A` and node `B` to node `Object` because both classes extend the root class `java.lang.Object`. (The ASG nodes for the methods of `Object` are omitted from the diagram.)

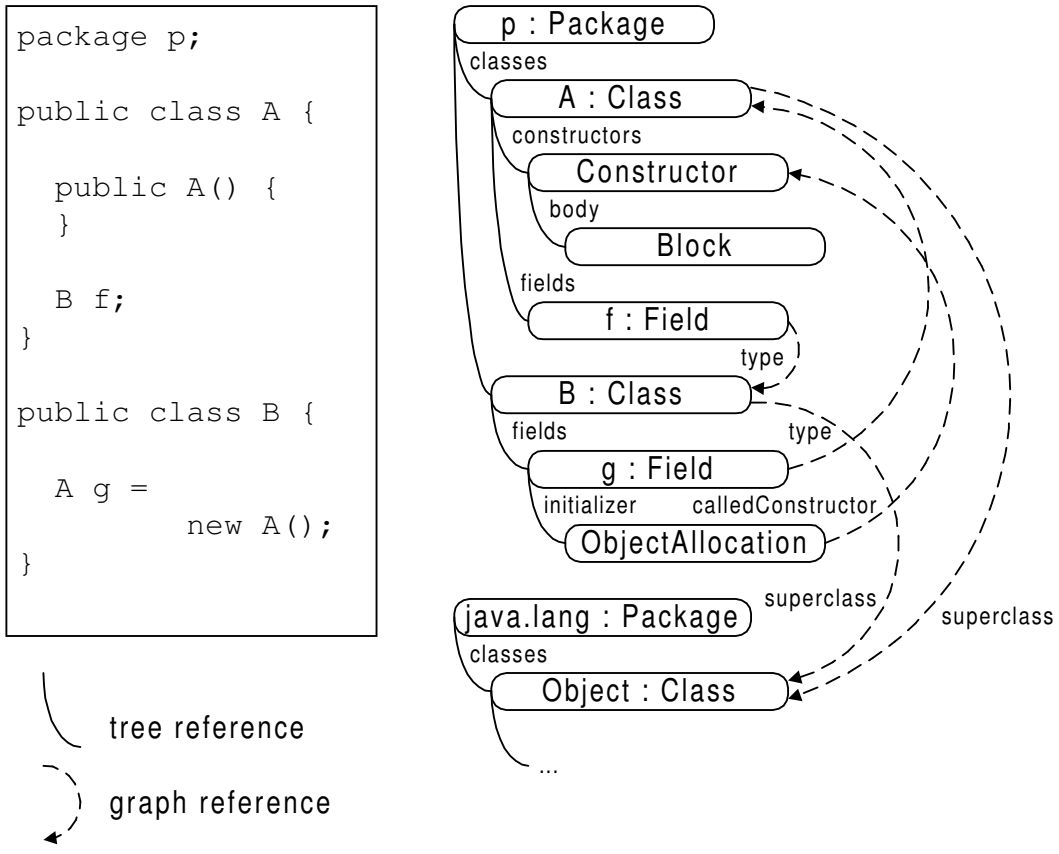

Figure 3.1: An example abstract semantics graph (ASG).

A complete description of the structure of the abstract semantics graphs built by CoffeeStrainer is given in Chapter 6, which documents all accessor methods that may be called on objects representing graph nodes.

3.1.2 A simple example

In this section, we present a simple CoffeeStrainer constraint to explain the principles on which CoffeeStrainer is based.

The main idea of CoffeeStrainer is to execute constraint-checking code, called *constraint methods*[1], at compile-time (or constraint checking time). The constraint methods, which are methods returning `boolean` values, are called by CoffeeStrainer during a single traversal of the ASG which is to be checked. They are written by programmers who want to enforce constraints regarding the classes, interfaces, methods or fields they have defined. Constraint methods are embedded in Javadoc comments [Gosling et al. 1996, Aitken 1996] just before the program element with wich they are associated. Normally, constraint methods have a single parameter, the ASG node which CoffeeStrainer is currently visiting during the ASG traversal. If a constraint method returns `false`, CoffeeStrainer reports a constraint violation on the currently visited ASG node.

We now turn to a very simple implementation of a static type definition constraint. The constraint requires that, for proper encapsulation, a class should declare all fields (instance variables) with `private` access only. This constraint leads to programs that are more maintainable, since the internal representation of an object's state can be changed without requiring changes to all users of that object. When fields are declared with `private` access, even subclasses of a class cannot access the fields directly, such that implementation changes in a base class need not lead to changes in derived classes.

We associate the constraint with the empty *marker interface* `AllFieldsArePrivate`, which makes it apply to all classes which implement the interface, i.e., to all classes which are marked with `AllFieldsArePrivate`. To highlight constraint methods, their names are set in **bold** face:

```
package coffeestrainer.examples;

/** Requires all fields of subtypes to be private.
 * @constraints
 * public boolean checkField(Field f) {
 *    rationale = "all fields must be private";
 *    return f.isPrivate();
 *  }
 */
public interface AllFieldsArePrivate {
}
```

As can be seen, constraint methods appear in a `constraints` paragraph of Javadoc comments just before the program element they are associated with. Javadoc comments are comments which start with "`/**`" and appear directly before a class, interface, method, or field. Each line of a Javadoc comment is stripped of any leading white space up to and including an optional asterisk character "`*`". They contain zero or more sentences of text which describe the program element they are associated with, followed by zero or more *tags*, which

[1] Note the difference between *method constraints*, i.e., constraints that are associated with a method of the base-level program (see Section 2.1 for their definition), and *constraint methods*, which are methods that implement constraints and are executed by CoffeeStrainer during constraint checking.

start with the character "@" at the beginning of a line and end before the next tag or at the end of the comment. CoffeeStrainer constraints are written as tags named "@constraints".

The return value of constraint methods is either true, if the constraint is satisfied, or false, if the constraint is violated. In case of a constraint violation, CoffeeStrainer reports it using the string stored in the special variable rationale.

Technically, CoffeeStrainer extracts tags named "@constraints" from Javadoc comments, and inserts the contained constraint methods into newly generated *constraint classes*, which are then compiled on the fly, and dynamically loaded into the CoffeeStrainer system. The constraint class generated for AllFieldsArePrivate is shown in Figure 3.2.

```
package constraints.coffeestrainer.examples;

import barat.reflect.*;

public class AllFieldsArePrivate
        extends coffeestrainer.InterfaceChecker {
  public static AllFieldsArePrivate thisInterface = null;
  public AllFieldsArePrivate() {
    if(thisInterface!=null)
      throw new RuntimeException("singleton only!");
    thisInterface = this;
  }
  public boolean checkField(Field f) {
    rationale = "all fields must be private";
    return f.isPrivate();
  }
}
```

Figure 3.2: Generated constraint class for AllFieldsArePrivate

Constraint classes are placed in a shadow package structure — because the interface AllFieldsArePrivate is contained in package coffeestrainer.examples, its corresponding constraint class is constraints.coffeestrainer.examples.AllFieldsArePrivate, i.e., it has the same name as the base-level class or interface, and it is placed in a package whose name is the original package name prefixed with "constraints.".

The superclass of the generated constraint class, coffeestrainer.InterfaceChecker, contains default implementations for all constraint methods. The default implementation is just to return true. Additionally, it defines the field rationale which is used by CoffeeStrainer for generating meaningful messages in the case of a constraint violation.

Each generated constraint class is an application of the singleton design pattern [Gamma et al. 1995]; it contains a static field thisInterface (or thisClass) which references the single instance of the constraint class. This instance is at the same time the representation of the base-level interface or class for which the constraint class was generated. In our example, the expression constraints.coffeestrainer.examples.AllFieldsArePrivate.thisInterface refers to the ASG node object that represents the interface coffeestrainer.examples.AllFieldsArePrivate at checking-time. For

certain constraints, being able to represent the corresponding ASG node is useful (see Figure 3.8 for an example).

The remaining content of the generated constraint class is copied from the `@constraints` tag. Note that this makes it possible to define helper methods as well which are not constraint methods themselves. These can be used to factor out code which is common to several constraint methods.

During traversal of the ASG, the constraint methods of the generated constraint classes are called according to the Visitor design pattern [Gamma et al. 1995]. Ignoring the default constraint methods (which return true) defined in `coffeestrainer.InterfaceChecker`, there is only one interesting constraint method, `checkField`. It is a definition constraint method because it begins with just "`check`". As it is associated with the interface `AllFieldsArePrivate`, it is invoked by CoffeeStrainer during the traversal of `AllFieldsArePrivate` itself and all subtypes of it, when an ASG node of type `Field` is encountered.

Consider the example class `Student`, whose source code and corresponding abstract semantics graph are shown in Figure 3.3.

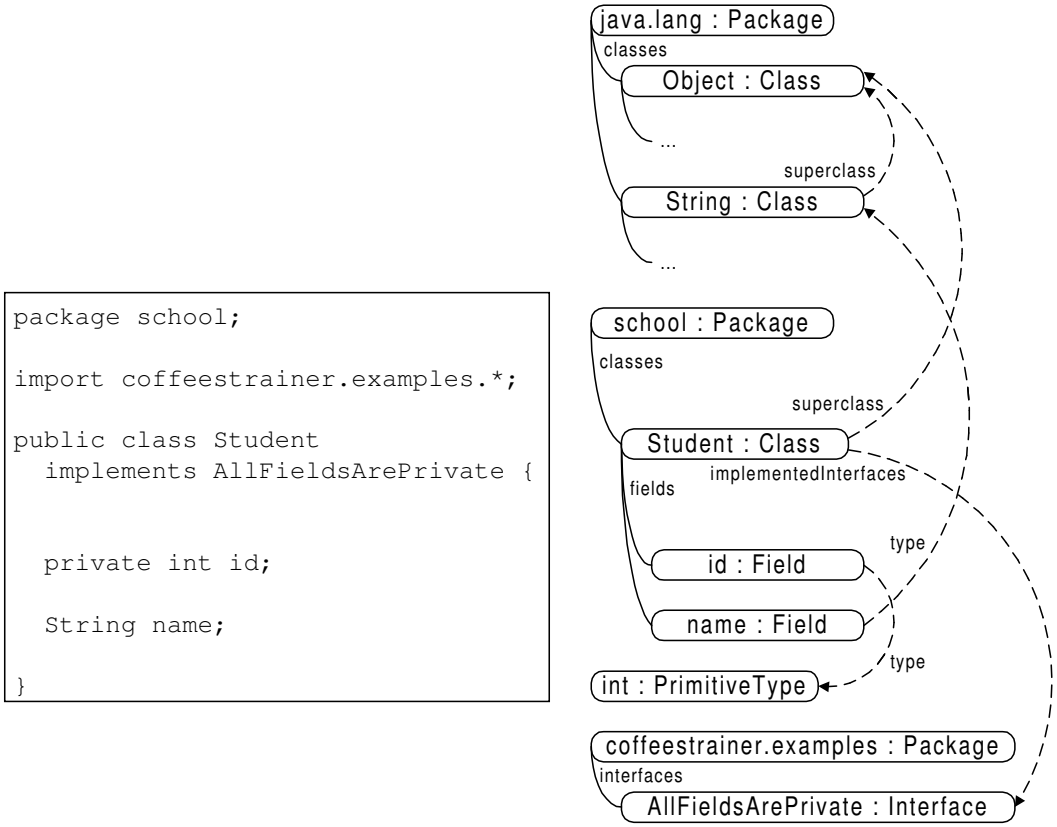

Figure 3.3: Student example

To the right of the source code, the ASG nodes and their types are shown. A node of type `Package` is at the root of the ASG. The only child of that node is a node of type `Class`, representing the class `Student`. This node then has two children of type `Field`, for `id` and `name`, respectively. Modifiers like `private`, `protected`, or `public` are stored as attributes of the

`Field` nodes and are not depicted in the diagram. From the nodes of the ASG of package `school`, there are four dotted arrows that represent information from name analysis. One arrow is from node `Student` to the node representing the class `java.lang.Object`, since `Student` inherits from `Object`. Another arrow is from node `Student` to the node representing `AllFieldsArePrivate`, because `Student` implements this interface. The third arrow is from node `name` to the node representing the class `java.lang.String`, a reference to the declared type of `name`. Finally, the fourth arrow is from node `id` to a predefined node representing the primitive type `int`. In the diagram, the ASG nodes for methods of `java.lang.Object` and for fields, methods, and constructors of `java.lang.String` are omitted.

The constraint defined in `AllFieldsArePrivate` applies to our sample class `Student` because Student implements `AllFieldsArePrivate`. Since this interface is empty, as seen from the perspective of a normal Java compiler, the subtype relationship can be declared without making additional changes to `Student`, such as e.g. implementing additional methods. When using CoffeeStrainer, it is a general technique to use empty interfaces for defining constraints, and then using these interfaces to mark classes for which the constraint should be enforced. Such marker interfaces are useful in other contexts as well; for example, the standard interface `Serializable` is such a marker interface that marks classes whose objects may be serialized and stored in a file or sent over the network.

The concrete steps taken by CoffeeStrainer for checking the example class `Student` are as follows:

- The ASG for `Student` is built and then augmented with name and type analysis information. During name and type analysis, classes and interfaces that are referenced by `Student` are parsed on demand.
- Eventually, the interface `AllFieldsArePrivate` is parsed, and the Javadoc comment is detected. From the code contained in the `@constraints` tag, the new constraint class `constraints.coffeestrainer.examples.AllFields-ArePrivate` (depicted in Figure 3.2) is generated, compiled on the fly and loaded dynamically.
- After all necessary files have been parsed, the actual checking is performed: For both nodes `id` and `name` of type `Field`, the method `checkField` will be called by the CoffeeStrainer framework, providing the ASG nodes representing `id` and `name` as an argument, respectively.
- The method `checkField` contains a rationale for the constraint. This rationale is passed to CoffeeStrainer by assigning it to the predefined string variable `rationale`. This step is optional, but makes the messages that are output for violations of the constraint easier to understand for the programmer.
- Then, if a node is found for which `checkField` returns `false`, which is the case for the node representing `name`, a field with package-local accessibility, a warning message will be generated. This warning message consists of the `rationale`, the name of the class which specified the constraint and information about the construct that violates the constraint (file name, line number, and source code):

```
$ java CoffeeStrainer.Main school.Student

AllFieldsArePrivate does not allow Field "name"
  (because all fields must be private)
  in file school/Student.java, line 5
```

3.1.3 The structure of CoffeeStrainer constraints

Generally, CoffeeStrainer constraint methods are executed at compile-time (or checking-time) and support the implementation of static constraints. The implementation of dynamic constraints is enabled by a special construct which allows to programmatically insert run-time checks into the code (see Section 3.3).

Constraint methods can be classified along three dimensions, namely, the kind of program element with which they are associated, whether they are definition constraint methods or usage constraint methods, and the ASG node type to which they apply:

- The kind of program element with which a constraint method is associated may be a class or interface, a method, or a field, leading to a *type constraint*, *method constraint*, or *field constraint*. Technically, constraint methods are embedded inside Javadoc comments that appear directly before the declaration of the program element - i.e., directly before a class or interface, directly before a method, or directly before a field.
- Constraint methods whose names start with just "`check`" implement definition constraints, and constraint methods whose names start with "`checkUseAt`" implement usage constraints.
- The ASG node type to which a constraint method applies is determined by the remaining name of the constraint method. For instance, the constraint method `checkField` is a definition constraint that applies to ASG nodes of type `Field`, and the constraint method `checkUseAtCast` is a usage constraint method that applies to ASG nodes of type `Cast`. Constraint methods have a single parameter whose type is the applicable ASG node type. For example, the declaration of the constraint method `checkField` is "`public boolean checkField(Field f);`", and the declaration of the constraint method `checkUseAtCast` is "`public boolean checkUseAtCast(Cast c);`".

Based on these three dimensions, CoffeeStrainer invokes constraint methods during a single traversal of the ASG that is to be checked, providing the currently visited ASG node as the argument:

- Type definition constraint methods are invoked for all ASG nodes of applicable type within the ASG of the type (class or interface) itself and the ASGs of all subtypes. Method definition constraint methods are invoked for all the ASG nodes of applicable type within the ASG of the method itself and the ASGs of all methods overriding that method, or implementing that method if it is an abstract method. Finally, field definition constraint methods are invoked for all ASG nodes of applicable type within the

ASG of the field's definition[2]. For example, a type definition constraint method called `checkCast`, which is associated with the interface `java.lang.Comparable`, is invoked for each ASG node of type `Cast` which appears in `java.lang.Comparable` itself or any of its subtypes. In contrast, if `checkCast` were associated with the method `compareTo` in the interface `Comparable`, it would be invoked for each ASG node of type `Cast` which appears in `compareTo` itself or in any of the methods in subtypes of `Comparable` that implement `compareTo`.

- Type usage constraint methods are invoked for all ASG nodes of applicable type within the whole program's ASG in which the type (class or interface) or one of its subtypes is used. Method usage constraint methods are invoked for all ASG nodes of applicable type within the whole program's ASG in which the method or one of its overriding (implementing) methods is used. Finally, field usage constraint methods are invoked for all ASG nodes of applicable type within the whole program's ASG in which the field is used. For example, a type usage constraint method called `checkUseAtCast`, which is associated with the interface `java.lang.Comparable`, is invoked for each ASG node of type `Cast` which uses `Comparable` or one of its subtypes as the casted type, regardless of whether this ASG node appears in `Comparable` or any of its subtypes. Because methods are not used by ASG nodes of type `Cast`, `checkUseAtCast` cannot be associated with the method `compareTo` in the interface `Comparable`. If another constraint method `checkUseAtInstanceMethodCall` was associated with `compareTo`, it would be invoked for every ASG node of type `InstanceMethodCall` which represents a call to `compareTo` itself or to any of the methods in subtypes of `Comparable` that implement `compareTo`.

Note that the scope of type constraint methods includes all subtypes of the type with which the constraint method is associated, and method constraint methods apply to the method with which they are associated and to all overriding methods. The reason for this is the object-oriented principle of substitutability [Liskov, Wing 1994]: Given a type T and a subtype S of T, objects of type S must be substitutable for objects of type T, and consequently, constraints that are checked on T must be checked on S as well.

This scheme makes checking constraints modular and well-suited for object-oriented programs: A type constraint (i.e., appearing directly before a class or interface definition) applies to the ASG of the class or interface itself and to all of its subtypes. For example, a constraint method `checkField` that appears in the `@constraints` tag of a Javadoc comment for a class `A`, applies to the ASG of `A` and to all ASGs of `A`'s subclasses. By defining additional constraint methods for a subclass `B` of `A`, the constraints that applied to `A` can be extended and refined for `B` and its subclasses. However, it is not possible to weaken constraints in subtypes; a constraint method that is placed in class `A` cannot be overridden in class `B`. If a constraint method of the same name is defined both in class `A` and class `B`, `A`'s constraint will apply to `A` and all subclasses of `A` (including `B`), and *additionally*, `B`'s constraint will apply to `B` and all subclasses of `B`.

Inheritance of normal Java classes is not reflected on the constraint class level, which is why overriding of constraint methods is not possible. Apart from the methodological argument

[2] In Java, there is no notion of overriding field definitions. If the base language allowed overriding of fields, field definition constraint methods should be invoked for the field itself and all overriding fields.

that constraints should only be strengthened by subtypes, there is a technical reason for this as well: The constraint methods defined for Java *interfaces* become methods of generated constraint *classes* — remember that only classes can contain concrete methods. Thus, because Java interfaces may have multiple superinterfaces, multiple implementation inheritance — not available in Java — would be needed at the level of the generated constraint classes.

3.2 Implementing static constraints

As has been explained earlier, CoffeeStrainer performs a tree traversal on the ASG of every compilation unit that is to be checked. In this section, we will describe the constraint methods that will be invoked during this traversal in more detail. For this, it is sufficient to explain which methods are called when visiting one particular node n of the ASG. The order of traversing ASG nodes corresponds to the occurrence of program elements in the source code for the compilation unit. However, the order should be irrelevant because it is considered good style for constraint methods to be free of side-effects[3].

In the following, we will make use of the following symbols, functions, and relations: T is the set of all types, i.e., the set of all classes and interfaces. For $t_1, t_2 \in T$, we write $t_1 \succ t_2$ iff t_1 is a supertype of t_2, and $t_1 \succeq t_2$ iff $t_1 \succ t_2 \vee t_1 = t_2$. M is the set of all methods, and we write $m_1 \succ m_2$ iff m_1 is overridden or implemented by m_2, and $m_1 \succeq m_2$ iff $m_1 \succ m_2 \vee m_1 = m_2$. Finally, F is the set of all fields.

The five subsections explain the constraint methods called for type definition constraints, method definition constraints, type usage constraints, method usage constraints, and field usage constraints. We do not cover field definition constraints, because they would only make sense in a language which supported overriding of fields in subclasses.

3.2.1 Implementing type definition constraints

Before explaining formally how type definition constraint methods are invoked by CoffeeStrainer, we motivate the scheme by means of a simple example. Figure 3.4 shows the type definition constraint methods that will be called during the traversal of the example ASG which was introduced in Figure 3.1 already. They are shown on the right of each ASG node. For each invocation of a constraint method, this ASG node is provided as the argument. The expressions `ccA`, `ccB`, and `ccO` denote the singleton instances of the generated constraint classes for `A`, `B`, and `java.lang.Object`, respectively. For example, the first two invocations `ccO.checkClass` and `ccA.checkClass` refer to the constraint method `checkClass` associated with the class `java.lang.Object` and the class `A`, respectively. For both invocations, the method argument is the ASG node representing class `A`.

The order of calling constraint methods is determined by a pre-order traversal of the ASG which corresponds to the order in which they occur in the source code.

[3]Note that CoffeeStrainer does not enforce that constraint methods be free of side-effects — doing so would only be possible at the cost of making constraint methods less expressive. Like explained in Section 3.1.2, they are copied verbatim from the `@constraints` tag of `Javadoc` comments.

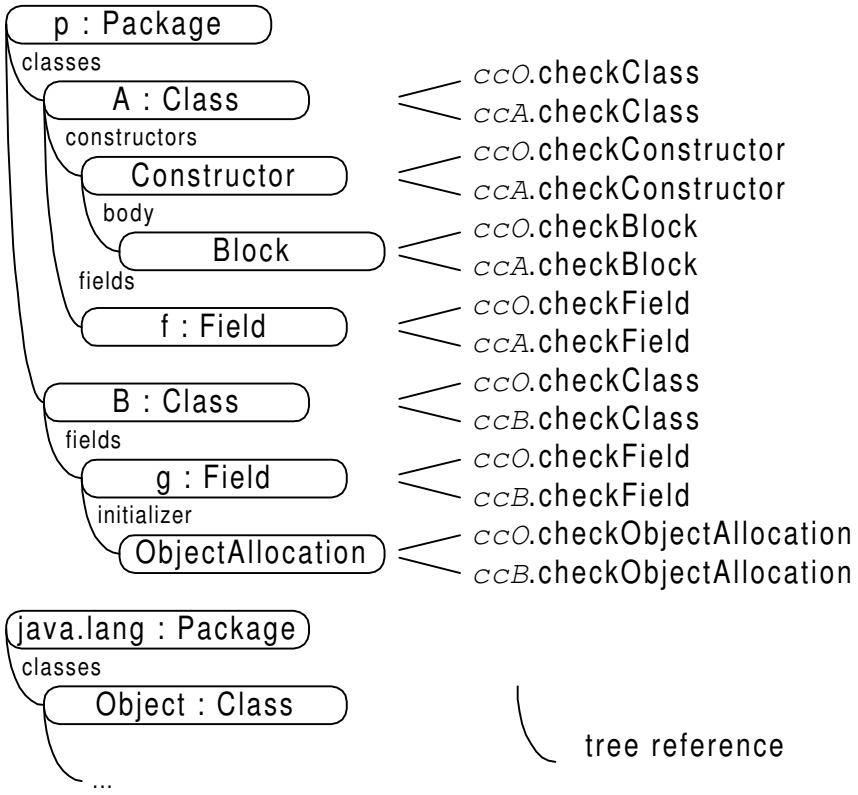

Figure 3.4: Definition constraint methods called for the example ASG

Note that all nodes in the ASG subtree for class `A` are checked by type definition constraint methods associated with class `A` itself, and by type definition constraint methods associated with class `java.lang.Object`, the supertype of `A`. Similarly, all nodes in the ASG subtree for class `B` are checked by type definition constraints associated with class `B` and again, by type definition constraint methods associated with class `java.lang.Object`, which also is the supertype of class `B`.

We now explain the constraint methods called for type definition constraints when visiting one ASG node n in a more formal way. Figures 3.5 and 3.6 show the signatures of all possible type definition constraint methods[4], starting with the ASG node types representing classes and interfaces with their constituents, followed by ASG node types representing statements, expressions, and other entities.

Let $t_n \in T$ be the type (class or interface) in which n is contained. Of all definition constraint methods — those starting with just "`check`" — only methods whose applicable ASG node type is the type of n will be called with n as argument. The applicable type of a constraint method is determined by the second part of its name, so if n is of type `Field`, potentially applicable definition constraint methods are called `checkField`, and if n is of type `Cast`, applicable definition constraint methods are called `checkCast`.

Let d be the common name of the applicable definition constraint methods. Then, for every type $t \succeq t_n$, a method d will be called on the generated constraint class for t with n as the argument.

For example, assume that the ASG node n is of type `Cast`, occurring in the type t_n=`java.lang.Integer`. The set of types $\{t \in T | t \succeq t_n\}$ for `java.lang.Integer` is {`java.lang.Integer`, `java.lang.Number`, `java.io.Serializable`, `java.lang.Object`}, and the common name d of all applicable definition constraint methods is `checkCast`. Thus, implementations of `checkCast` associated with `java.lang.Integer`, `java.lang.Number`, `java.io.Serializable`, and `java.lang.Object`, if existent, will be called for checking type definition constraints for the ASG node n.

We can now proceed to implement the example type definition constraint for `java.io.Serializable`, which requires that non-serializable superclasses of serializable classes have an accessible no-arg constructor (see Section 2.2.1 on page 24 for a detailed explanation of this constraint):

```
/**
 *@constraints
 * public boolean checkClass(Class c) {
 *   rationale = "non-serializable superclass needs accessible no-arg constructor";
 *   if c.getSuperclass().isAssignableTo(thisInterface) return true;
 *   Constructor ctor =c.getSuperclass().getConstructor(new AType[]);
 *   return ctor!=null && ctor.isAccessibleFrom(c);
 * }
 */
```

[4]Because the argument types of all these methods are different, they could all be called `check` without any suffix, relying on overloading to distinguish between them. We chose not to use overloading for two reasons: First, code making use of overloading is more difficult to read because one needs to consider argument types *and* names to distinguish between methods, and second, this naming scheme fits better with the naming scheme used for usage constraint methods, which are introduced in Section 3.2.3.

```
public boolean checkClass(Class o);
public boolean checkInterface(Interface o);
public boolean checkAbstractMethod(AbstractMethod o);
public boolean checkConcreteMethod(ConcreteMethod o);
public boolean checkConstructor(Constructor o);
public boolean checkField(Field o);
public boolean checkBlock(Block o);
public boolean checkBreak(Break o);
public boolean checkCatch(Catch o);
public boolean checkContinue(Continue o);
public boolean checkDo(Do o);
public boolean checkEmptyStatement(EmptyStatement o);
public boolean checkExpressionStatement(ExpressionStatement o);
public boolean checkFinally(Finally o);
public boolean checkFor(For o);
public boolean checkIf(If o);
public boolean checkReturn(Return o);
public boolean checkSwitch(Switch o);
public boolean checkSynchronized(Synchronized o);
public boolean checkThrow(Throw o);
public boolean checkTry(Try o);
public boolean checkUserTypeDeclaration(UserTypeDeclaration o);
public boolean checkVariableDeclaration(VariableDeclaration o);
public boolean checkWhile(While o);
public boolean checkCaseBranch(CaseBranch o);
public boolean checkConstructorCall(ConstructorCall o);
public boolean checkDefaultBranch(DefaultBranch o);
public boolean checkForInitDeclaration(ForInitDeclaration o);
public boolean checkForInitExpression(ForInitExpression o);
public boolean checkLocalVariable(LocalVariable o);
public boolean checkParameter(Parameter o);
```

Figure 3.5: Names and signatures of definition constraint methods (Part I)

```
public boolean checkAnonymousAllocation(AnonymousAllocation o);
public boolean checkArrayAccess(ArrayAccess o);
public boolean checkArrayAllocation(ArrayAllocation o);
public boolean checkArrayInitializer(ArrayInitializer o);
public boolean checkArrayLengthAccess(ArrayLengthAccess o);
public boolean checkAssignment(Assignment o);
public boolean checkBinaryOperation(BinaryOperation o);
public boolean checkCast(Cast o);
public boolean checkConditional(Conditional o);
public boolean checkInstanceFieldAccess(InstanceFieldAccess o);
public boolean checkInstanceof(Instanceof o);
public boolean checkInstanceMethodCall(InstanceMethodCall o);
public boolean checkLiteral(Literal o);
public boolean checkObjectAllocation(ObjectAllocation o);
public boolean checkParenExpression(ParenExpression o);
public boolean checkStaticFieldAccess(StaticFieldAccess o);
public boolean checkStaticMethodCall(StaticMethodCall o);
public boolean checkThis(This o);
public boolean checkSuper(Super o);
public boolean checkUnaryOperation(UnaryOperation o);
public boolean checkVariableAccess(VariableAccess o);
```

Figure 3.6: Names and signatures of definition constraint methods (Part II)

```
public interface Serializable {
}
```

The definition constraint method `checkClass` will be called for each ASG node representing a class that is a subtype of `Serializable`. Remember that the special variable `thisInterface` references the ASG node representing the associated interface `Serializable`. Only if the superclass of the checked class is not a subtype of `Serializable`, one needs to check that this superclass contains an accessible constructor with no arguments. Therefore, the constraint implementation returns `true` if the argument class's superclass is assignable to `thisInterface`, the ASG node for the interface `Serializable`. If this is not the case, the superclass is searched for a constructor with no arguments. The method `getConstructor`, defined for ASG node objects representing classes, returns the ASG node object for the constructor whose argument types are given as arguments of `getConstructor`, or `null` if no such constructor exists. The result of the constraint method is `true` only if the constructor exists and if it is accessible from the class `c` which is checked by the constraint method.

3.2.2 Implementing method definition constraints

In this section, we consider the constraint methods called for method definition constraints when visiting one ASG node n.

Let $m_n \in M$ be the method in which n is contained. As explained in the previous section, the name of potentially applicable definition constraint methods is "`check`" followed by the

name of the ASG node type of n. For example, if the ASG node type of n is `Assignment`, the applicable definition constraint methods are called `checkAssignment`.

From the list of possible definition constraint methods, shown in Figures 3.5 and 3.6, only a few are irrelevant for method definition constraints, namely, `checkClass`, `checkInterface`, `checkConstructor`, and `checkField`. All other definition constraint methods can appear in method definition constraints, as the applicable ASG node type may be part of a method's ASG.

Let d be the common name of the applicable definition constraint methods. Then, for every method $m \succeq m_n$, a constraint method d will be called. In other words, definition constraints that are associated with a method, i.e., that appear in Javadoc comments just before a method definition, apply to the method with which they are associated and to all overriding methods.

The constraint classes generated for methods with associated constraints are similar to the constraint classes generated for classes and interfaces. Instead of `thisClass` or `thisInterface`, they have a field `thisMethod` which can be used in the constraint methods to refer to the method with which they are associated. They are realized as static inner classes of the constraint class generated for their containing class or interface. This enables the constraint programmer to reuse helper methods across constraints associated with different methods in the same class or interface. For the following example of a method definition constraint, we will show the generated constraint class.

We can now proceed to implement the method definition constraint of Section 2.2.2. The constraint required that methods overriding `addImpl` of `java.awt.Container` should call `super.addImpl`. For brevity, we assume that a restrictive interpretation of the rule was intended, which requires that the first statement of the method be the call to `super`. The ASG structure required by this constraint is exemplified in Figure 3.7. It shows a part of the ASG of class `Container` (only one method ASG node is shown) and the ASG of an example subclass called `Holder` in package `user`. The implementation of `addImpl` in `Holder` includes the call to the overridden method as its first statement. The source code is indented in a non-standard way to make it easy to associate the ASG nodes on the right with the source code part on the left.

A `ConcreteMethod` node contains a node of type `Block`, whereas a node of type `AbstractMethod` does not. The pseudo-variable `super` is represented by a node of type `Super`.

The desired constraint can be written as follows:

```
public abstract class Container {
  /**
   *@constraints
   * static boolean callsMethod(AStatement s, AMethod m) {
   *   if(!(s instanceof ExpressionStatement)) return false;
   *   AExpression e=((ExpressionStatement)s).getExpression();
   *   if(!(e instanceof InstanceMethodCall)) return false;
   *   AMethod called=(InstanceMethodCall)e).getCalledMethod();
   *   return called == m;
   * }
```

```
package java.awt;

public class Container {

  public void addImpl(Component c) {


    ...
  }
  ...
}
package user;
import java.awt.*;

public class Holder extends Container {

  public void addImpl(
                        Component c) {




    super.addImpl(




                     c);

    ...
  }
}
```

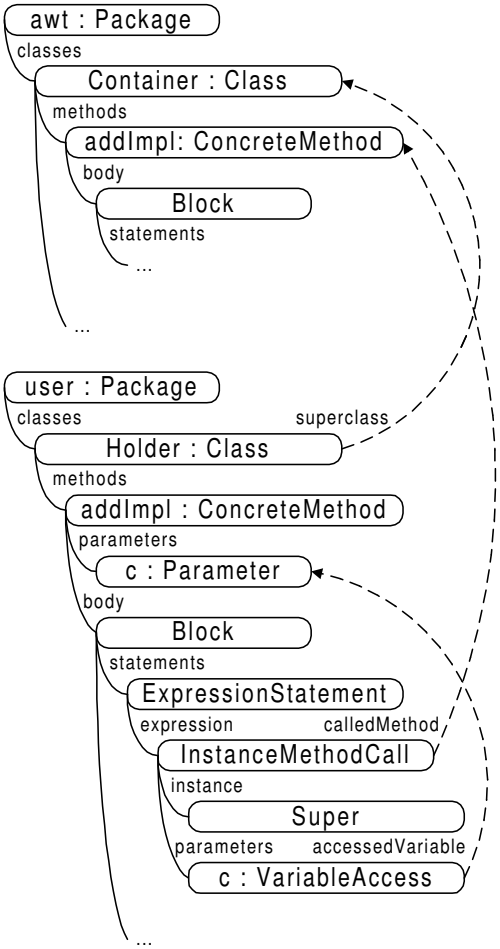

Figure 3.7: The ASG of class `Container` and a subclass

```
   * public boolean checkConcreteMethod(ConcreteMethod m) {
   *   if(m==thisMethod) return true;
   *   rationale = "when overriding addImpl, " +
   *               "super.addImpl() must be a top-level statement";
   *   StatementList sl = m.getBody().getStatements();
   *   return sl.size()!=0 && callsMethod(sl.get(0),
   *                                      m.getOverriddenMethod());
   * }
   */
  public void addImpl(Component c) {
    ...
  }
}
```

The method definition constraint method `checkConcreteMethod` is called for `addImpl` in class `Container` and for all methods that override it in subclasses. When it is called on an overriding method (i.e., `m!=thisMethod`), the constraint method checks that the first statement is a call to the overridden method.

The generated constraint class is shown in Figure 3.8. Constraint classes generated for methods are static inner classes of the constraint classes generated for their containing class or interface. The names of constraint classes generated for methods are "`Method_n`", where `n` is the index of the method in its class. In our example, the constraint class generated for `addImpl` is called `Container.Method_12` because `addImpl` has index 12 of all methods in `java.awt.Container` (constructors are counted as methods as well).

3.2.3 Implementing type usage constraints

Types (classes and interfaces) can be used in two ways: either by explicitly naming the type, as for example in field declarations, `instanceof` expressions, and object allocations, etc., or in the form of values of the type, i.e., if an expression has the type as its static type. In the former case, the ASG node in which the type name occurs is the main point of interest. In the latter case, it is not the expression itself, but the parent node of the expression which is the main point of interest.

In this section, we again use a simple example before explaining formally how type usage constraint methods are invoked by CoffeeStrainer. Figure 3.9 shows the usage constraint methods that will be called during the traversal of the example ASG which was introduced in Figure 3.1. They are shown on the right of the ASG node which is provided as the argument of the constraint method invocation. The expressions *ccA*, *ccB*, and *ccO* denote the singleton instances of the generated constraint classes for `A`, `B`, and `java.lang.Object`, respectively. For example, the second and third invocations `ccO.checkUseAtField` and `ccB.checkUseAtField` refer to the constraint method `checkUseAtField` associated with the class `java.lang.Object` and the class `B`, respectively. For both invocations, the method argument is the ASG node representing the field `f` of class `A`. Note that the third invocation (`ccB.checkUseAtField`) calls a usage constraint method associated with class `B` even though the field `f` which is provided as the argument appears in the ASG part for class `A`.

```
package constraints.java.awt;

import barat.reflect.*;

public class Container extends coffeestrainer.ClassChecker {

  ... // code generated for constraint class Container

  public static class Method_12
    extends coffeestrainer.MethodChecker {

    public static Container_method_12 thisMethod = null;
    public Container_method_12() {
      if(thisMethod!=null)
        throw new RuntimeException("singleton only!");
      thisMethod = this;
    }
    static boolean callsMethod(AStatement s, AMethod m) {
      if(!(s instanceof ExpressionStatement)) return false;
      AExpression e=((ExpressionStatement)s).getExpression();
      if(!(e instanceof InstanceMethodCall)) return false;
      AMethod called=(InstanceMethodCall)e).getCalledMethod();
      return called == m;
    }
    public boolean checkConcreteMethod(ConcreteMethod m) {
      if(m==thisMethod) return true;
      rationale = "when overriding addImpl, " +
                  "super.addImpl() must be a top-level statement";
      StatementList sl = m.getBody().getStatements();
      return sl.size()!=0 && callsMethod(sl.get(0),
                                         m.getOverriddenMethod());
    }
  }
}
```

Figure 3.8: Constraint class generated for a method constraint

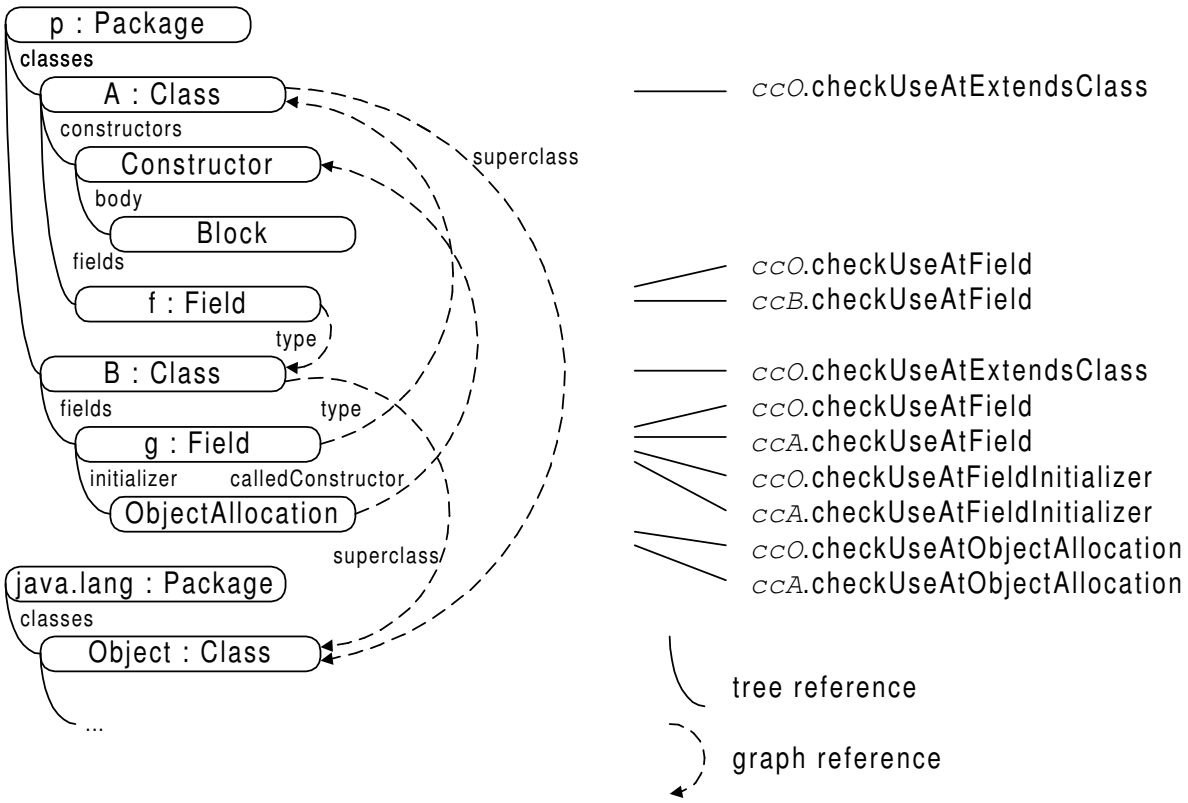

Figure 3.9: Usage constraint methods called for the example ASG

The last six usage constraint method calls in Figure 3.9 concern three different uses of `A` (and therefore, because `A`'s superclass is `Object`, also of `Object`): first, as the declared type of field `g`; second, as the static type of the field initializer expression (which happens to be an object allocation expression), and third, as the class referred to by name in the object allocation expression. The methods `checkUseAtField` and `checkUseAtFieldInitializer` take the `Field` object as their argument, and the method `checkUseAtObjectAllocation` takes the `ObjectAllocation` object as its argument.

Note that usage constraint methods apply to every use of a certain type in a program that is checked by CoffeeStrainer: While checking class `A`, one usage constraint method of `B` is called because `B` is used once in class `A`, and two usage constraint methods of `Object` are called because `Object` is used twice in `A` (directly, as `A`'s superclass, and indirectly, as `B`'s superclass because `B` is used in the field declaration). Similarly, three usage constraint methods defined in class `A` are called while checking class `B`, because class `B` uses `A` in three different ways, and four usage constraint methods defined for `Object` are called, because class `B` uses `Object` (directly or indirectly) in four different ways.

We now explain the constraint methods called for type usage constraints when visiting one ASG node n in a more formal way. Figure 3.10 shows the signatures of all possible type usage constraint methods in alphabetical order.

First, we consider the case that n names other (class or interface) types. In this case, the common name d of the potentially applicable usage constraint methods is "`checkUseAt`" followed by the name of the ASG node type of n. For example, if the ASG node type of n is `ObjectAllocation`, the name is `checkUseAtObjectAllocation`. Furthermore, let $T_n \subseteq T$ be the set of types which are named explicitly by n. Usually, this set is either empty or it contains a single element, because most Java program elements explicitly reference at most one user-defined type. Then, the method d with n as the argument will be called for the constraint classes generated from all $t \in T$ for which there is a $t' \in T_n$ for which $t \succeq t'$, i.e., d will be called for all constraint classes generated for types in T_n and those generated for their supertypes.

Second, we consider the case that n uses a type by having an expression node as a child. Let C be the set of pairs (t_e, s_e) describing child expression nodes of n where $t \in T$ is the expression's static type and s_e is a String which differentiates between multiple children expression nodes[5]. Then, for each $(t_e, s_e) \in C$, the common name d of the potentially applicable usage constraint methods is "`checkUseAt`" followed by the name of the ASG node type of n and the String s_e. For example, if the ASG node type of n is `Assignment`, which has two child expression nodes, the names of the potentially applicable type usage constraint methods are `checkUseAtAssignmentLValue` and `checkUseAtAssignmentOperand`. This method d will be called with n as the argument for the constraint classes generated from all $t \in T$ for which $t \succeq t_e$. In other words, for each expression child node of n whose static type is t_e, usage constraint methods will be called whenever they are defined in t_e or one of its supertypes.

Note that the list of possible usage constraint methods shown in Figure 3.10 contains one exception to the naming scheme described in the previous paragraph: The method `checkUse-`

[5]For example, ASG nodes of type `Assignment` have two children expression nodes: the l-value and the right-hand side of the assignment. In this case, the differentiating strings would be "`LValue`" and "`Operand`". If there is only one child expression node, the differentiating string is defined to be the empty string.

```
public boolean checkUseAtArrayAllocation(ArrayAllocation where);
public boolean checkUseAtAssignmentLValue(Assignment where);
public boolean checkUseAtAssignmentOperand(Assignment where);
public boolean checkUseAtBinaryOperation(BinaryOperation where);
public boolean checkUseAtCast(Cast where);
public boolean checkUseAtCastOperand(Cast where);
public boolean checkUseAtCatch(Catch where);
public boolean checkUseAtConditionalIfTrue(Conditional where);
public boolean checkUseAtConditionalIfFalse(Conditional where);
public boolean checkUseAtField(Field where);
public boolean checkUseAtFieldInitializer(Field where);
public boolean checkUseAtInstanceFieldAccess(InstanceFieldAccess where);
public boolean checkUseAtInstanceMethodCall(InstanceMethodCall where);
public boolean checkUseAtInstanceof(Instanceof where);
public boolean checkUseAtLocalVariable(LocalVariable where);
public boolean checkUseAtLocalVariableInitializer(LocalVariable where);
public boolean checkUseAtMethodCallParameter(int index, AMethodCall where);
public boolean checkUseAtMethodParameter(int index, AMethod where);
public boolean checkUseAtMethodResult(AMethod where);
public boolean checkUseAtObjectAllocation(ObjectAllocation where);
public boolean checkUseAtReturn(Return where);
public boolean checkUseAtStaticFieldAccess(StaticFieldAccess where);
public boolean checkUseAtStaticMethodCall(StaticMethodCall where);
public boolean checkUseAtSuper(Super where);
public boolean checkUseAtSynchronized(Synchronized where);
public boolean checkUseAtThis(This where);
public boolean checkUseAtThrow(Throw where);
public boolean checkUseAtThrows(AMethod where);
public boolean checkUseAtVariableAccess(VariableAccess where);
```

Figure 3.10: Names and signatures of usage constraint methods

AtMethodCallParameter has an int parameter that distinguishes between the different child nodes of a MethodCall node, instead of several methods with different names checkUseAtMethodCallParameter1, checkUseAtMethodCallParameter2, ... that encode the parameter index in the methods' names. For usage constraint methods one could not choose a single name checkUseAt because the parameter type does not distinguish between, e.g., checkUseAssignmentLValue and checkUseAtAssignment-Operand. Thus, as already mentioned, overloading is not used for definition constraint methods either in order to keep a single consistent naming scheme.

We can now proceed to implement the example type usage constraint described in Section 2.2.3, which requires that synchronization for Writer objects should be performed on the object in the field lock of class Writer instead of on a Writer object itself using synchronized methods or explicit synchronized blocks. As a first step, it is easy to disallow synchronized methods in subclasses of Writer by writing the following definition constraint method:

```
/**
 *@constraints
 * public boolean checkConcreteMethod(ConcreteMethod m) {
 *   rationale = "use field 'lock' instead of synchronized methods";
 *   return !m.isSynchronized();
 * }
 */
public class Writer {
}
```

The class definition constraint method checkConcreteMethod will be called for all concrete methods of Writer and subtypes of Writer. It returns true only if the method is not synchronized. Note that in Java, the synchronized property of methods is not inherited by overriding methods.

Adding a usage constraint method, one can do even better, and disallow synchronized statements that operate on objects of type Writer:

```
/**
 *@constraints
 * public boolean checkConcreteMethod(ConcreteMethod m) {
 *   rationale = "use field 'lock' instead of synchronized methods";
 *   return !m.isSynchronized();
 * }
 * public boolean checkUseAtSynchronized(Synchronized s) {
 *   rationale = "synchronize on field 'lock' instead of on Writer objects";
 *   return false;
 * }
 */
public class Writer {
}
```

The class usage constraint method checkUseAtSynchronized will be called for all ASG nodes of type Synchronized which synchronize on an object of type Writer. Such uses of Writer objects are not allowed; thus, the constraint method returns false.

3.2.4 Implementing method usage constraints

In this section, we consider the constraint methods called for method usage constraints when visiting one ASG node n.

Unlike for definition constraints, there are not many possible method usage constraint methods because there is only one way methods can be used — calling them. Thus, there are only two possible method usage constraint methods, shown in Figure 3.11, for constraining calls of instance method calls and static method calls, respectively.

Thus, method usage constraint methods will be called for n only if the ASG node type of n is `InstanceMethodCall` or `StaticMethodCall`. In the first case, the possibly applicable constraint methods are called `checkUseAtInstanceMethodCall`, and in the second case the possibly applicable constraint methods are called `checkUseAtStaticMethodCall`. Again, we define d to be the name of the applicable constraint methods.

Let $m_u \in M$ be the method which is called by n. Then, for every constraint class generated from a method $m \succeq m_u$, a constraint method d will be called with n as its argument.

```
public boolean checkUseAtInstanceMethodCall(InstanceMethodCall where);
public boolean checkUseAtStaticMethodCall(StaticMethodCall where);
```

Figure 3.11: Names and signatures of method usage constraint methods

Using method usage constraint methods, we can implement the constraint described in Section 2.2.4, which requires that the methods `addNotify` and `removeNotify` in class `java.awt.Component` should only be called from classes in the package `java.awt.peer` that forms the platform-dependent implementation of the AWT classes:

```
package java.awt;

/**
 *@constraints
 * private static Package peerPackage = Barat.getPackage("java.awt.peer");
 * static boolean callIsAllowed(AMethodCall mc) {
 *   return mc.containing(Package.class) == peerPackage;
 * }
 */
public class Component {

  ...

  /**
   *@constraints
   * public boolean checkUseAtInstanceMethodCall(InstanceMethodCall c) {
   *   rationale = "this method should not be called by application code";
   *   return callIsAllowed(c);
   * }
   */
  public void addNotify() {
    ...
```

```
  }
  /**
   *@constraints
   * public boolean checkUseAtInstanceMethodCall(InstanceMethodCall c) {
   *   rationale = "this method should not be called by application code";
   *   return callIsAllowed(c);
   * }
   */
  public void removeNotify() {
    ...
  }
}
```

The method usage constraint methods for `addNotify` and `removeNotify` will be called for every ASG node of type `InstanceMethodCall` which represents a call to the respective method or one of its overriding methods. They return `true` only if the call is allowed, a decision that is based on the context of the checked ASG nodes: Only ASG nodes that are part of the package `java.awt.peer` may perform the calls. Note the helper method `callIsAllowed`, which is placed in the constraint class generated for `java.awt.Container` and thus is accessible from both method constraints. The implementation of `callIsAllowed` uses the expression `mc.containing(Package.class)` to retrieve the ASG node of type `Package` which contains the method call `mc`.

3.2.5 Implementing field usage constraints

In this section, we consider the constraint methods called for field usage constraints when visiting one ASG node n.

Unlike field definition constraints, field usage constraints have a non-trivial scope, i.e. all ASG nodes that access a particular field. Accordingly, there are two possible field usage constraint methods, shown in Figure 3.12, for constraining accesses to instance fields and static fields, respectively.

Thus, field usage constraint methods will be called for n only if the ASG node type of n is `InstanceFieldAccess` or `StaticFieldAccess`. In the first case, the possibly applicable constraint methods are called `checkUseAtInstanceFieldAccess`, and in the second case the possibly applicable constraint methods are called `checkUseAtStaticFieldAccess`. We define d to be the name of the applicable constraint methods.

Let $f_u \in F$ be the field which is accessed by n. Then, the constraint method d of a constraint class generated for f_u will be called with n as its argument.

```
public boolean checkUseAtInstanceFieldAccess(InstanceFieldAccess where);
public boolean checkUseAtStaticFieldAccess(StaticFieldAccess where);
```

Figure 3.12: Names and signatures of field usage constraint methods

Using field usage constraints, we can implement the example constraint from Section 2.2.5, which is associated with the field `services` in class `java.beans.beancontext.BeanContextSupport`. The constraint requires that "all accesses to the `protected transient`

HashMap services field should be synchronized on that object." For simplicity, we check a sufficient condition for this constraint. We require that all usages of this field are contained in synchronized methods of the class java.beans.beancontext.BeanContextSupport or one of its subtypes:

```
package java.beans.beancontext;

public class BeanContextSupport {

  /**
   *@constraints
   * public boolean checkUseAtInstanceFieldAccess(InstanceFieldAccess o) {
   *   rationale = "accesses to services should be in synchronized methods";
   *   AMethod m = o.containingMethod();
   *   return    m!=null
   *          && m.containingClass().isAssignableTo(thisClass)
   *          && m.isSynchronized();
   * }
   */
  protected transient java.util.HashMap services;

  ...
}
```

The method checkUseAtInstanceFieldAccess will be called for all ASG nodes representing field accesses to services, regardless of the class they are contained in. The check m!=null is required because field accesses could have no containing method, for example, when they are part of the initializer expression of another field.

3.3 Implementing dynamic constraints

So far, only static constraints have been explained. This section explains how constraints can be implemented which cannot be checked at compile-time at all, or which can only be checked partly at compile-time.

A *dynamic constraint* is a constraint which can only be checked at run-time, i.e., which depends on run-time information such as the values of variables, or the actual control flow within the program. Unlike static constraints, which are boolean expressions that are evaluated once for each ASG node they apply to, dynamic constraints are boolean expressions that should be evaluated repeatedly in the course of running a program. However, like static constraints, dynamic constraints in CoffeeStrainer are associated with ASG nodes: For all ASG nodes which represent an entity that may be executed or evaluated at run-time, one may think of a dynamic constraint that is checked just before or just after executing or evaluating the run-time entity.

One can think of dynamic constraints as assertions [Floyd 1967, Floyd 1971] that can be inserted into certain parts of a program automatically. For example, method postconditions [Meyer 1992, Meyer 1997] are dynamic constraints that should be checked just after calculating the method's return value and just before returning control to the methods's caller. As

another example, one may wish to check just before executing certain field access expressions that the current instance has the necessary run-time privileges to access that field.

Interestingly, only one additional primitive method is needed to support dynamic constraints. This method, which always returns `true`, may be called from within constraint methods and is called `atRuntime`. It has three arguments: an ASG node object `n` and two character strings `pre` and `post`. The method `atRuntime(n, pre, post)`, when called on an ASG node `n` representing an executable entity (i.e., a method, a statement, or an expression), adds, as a side-effect, the dynamic check contained in `pre` just before each execution of `n` at run-time and the dynamic check contained in `post` just after each execution of `n` at run-time. The Strings `pre` and `post` are parsed as boolean expressions and type-checked in the context of `n`. They have access to all run-time entities accessible from that context, such as local variables, parameters, the current instance (`this`), etc.

For example, consider the following usage constraint method, which is a made-up example not taken from the Java standard classes. In Section 3.3.1, we will explain how method preconditions, which are the majority of dynamic constraint that have been found, can be implemented with CoffeeStrainer in a general way.

```
/**
 *@constraints
 * public boolean checkUseAtInstanceMethodCall(InstanceMethodCall c) {
 *   rationale = "caller needs valid certificate";
 *   return atRuntime(c, "Certificates.hasValidCertificate(this)", null);
 * }
 */
public class SecureObject {
}
```

This constraint method is called by CoffeeStrainer for every instance method that is called on an object of type `SecureObject`. There are two cases which can be distinguished:

- The dynamic constraint expression cannot be parsed: This might be the case if no current instance (`this`) is accessible in the context of the ASG node `c`[6]. In this case, `atRuntime` returns `false` and CoffeeStrainer statically reports a violation of the usage constraint method, notifying the user about the parsing problem.
- The dynamic constraint is enabled and parsing of the constraint expression succeeds: In this case, `atRuntime` returns `true`, i.e., there is no constraint violation which is reported statically. The checking of the constraint expression is inserted just before each method call on a method of `SecureObject`.

The constraint expression is inserted as follows: Assume that a method of `SecureObject` is called as in the following source code snippet:

[6] It is good style not to rely on this feature. The example constraint could be improved by adding a static check that ensures that the context of `c` includes a current instance (`this`) as follows:

```
return c.containingMethod()!=null
&& !c.containingMethod.isStatic()
&& atRuntime(...);
```

```
SecureObject secureObject = new SecureObject();
secureObject.someMethod();
```

Then, the resulting code after inserting the dynamic constraint expression looks like this (the inserted code is set in bold face):

```
(1) SecureObject secureObject = new SecureObject();
(2) if(!Certificates.hasValidCertificate(this))
(3)  throw new DynamicConstraintViolation(
(4)               "SecureObject",
(5)               "SecureObjectUser.java, line 3",
(6)               "caller needs valid certificate");
(7) secureObject.someMethod();
```

The inserted code checks the constraint (line 2) and causes an exception to be thrown if the constraint expression is `false` (line 3). The exception object is provided with information about the type which defined the constraint (line 4), file name and line number information (line 5) and the rationale for the constraint (line 6). Then, processing proceeds normally. If there had been a third argument to the call of `atRuntime`, an additional constraint check would have been inserted after the method call.

Note that the example constraint is interesting because unlike pre- and postconditions, it is checked in the caller's context instead of the callee's.

Often, dynamic constraints are to be checked either before the execution of a method, statement or expression, or after the execution, but not both before and after the execution at the same time. Therefore, CoffeeStrainer provides two convenience method `preRuntime(n, pre)` and `postRuntime(n, post)` which call `atRuntime(n, pre, null)` and `atRuntime(n, null, post)`, respectively.

The dynamic constraint that is checked *after* each evaluation of an expression can refer to the evaluated expression's value using the special variable `$value`. It is a parse error if a dynamic constraint refers to `$value` if the node to which it applies does not represent an expression. Consider the following constraint method, which is a refinement of the example given above:

```
/**
 *@constraints
 * public boolean checkUseAtInstanceMethodCall(InstanceMethodCall c) {
 *   rationale = "caller needs valid certificate for callee";
 *   return postRuntime(c.getInstance(),
 *                      "Certificates.hasValidCertificate(this, $value)");
 * }
 */
public class SecureObject {
}
```

The example is modified in three aspects: First, it uses the convenience method `postRuntime` rather than `atRuntime`. Second, the ASG node for which `postRuntime` inserts a

runtime check is not the method call `c` itself, but it is the ASG node `c.getInstance()` which represents the instance expression that yields the object on which the method call is peformed. Thus, it is possible to refer to that expression's value in the dynamic constraint as `$value`. Note that the expression "`c.getInstance()`" is evaluated at constraint checking time and not at run-time. Third, the constraint string now calls a hypothetical method `hasValidCertificate(caller, callee)` that checks whether the object referred to by `caller` has a valid certificate which allows it to call methods on the object referred to by `callee`. The parameter `callee` in this case is bound to the value of the instance expression, which can be referred to as `$value`. Again, assume that a method on an object of class `SecureClass` is called from the following code snippet:

```
SecureObject secureObject = new SecureObject();
secureObject.someMethod();
```

The resulting code after inserting the runtime check then looks as follows (inserted code set in bold face):

```
(1) SecureObject secureObject = new SecureObject();
(2) SecureObject $value = secureObject;
(3) if(!Certificates.hasValidCertificate(this, $value))
(4)  throw new DynamicConstraintViolation(
(5)               "SecureObject",
(6)               "SecureObjectUser.java, line 3",
(7)               "caller needs valid certificate for callee");
(8) $value.someMethod();
```

The inserted code assigns the value of the instance expression to `$value` (line 2), checks the constraint (line 3) and causes an exception to be thrown if the constraint expression is `false` (line 4). The exception object is provided with information about the type which defined the constraint (line 5), file name and line number information (line 6) and the rationale for the constraint (line 7). Then, the method call is performed on `$value` (line 8), and processing proceeds normally. To avoid name clashes, the actual implementation of CoffeeStrainer changes the name of `$value` to a unique variable name for every insertion of constraint checking code.

3.3.1 Implementing Eiffel-style preconditions using tags

In Java, classes, interfaces, methods, and fields can be annotated with Javadoc comments. CoffeeStrainer parses these comments and not only generates constraint classes from `@constraint` tags, it also makes all tags in Javadoc comments accessible on the level of ASG node objects. Tags are needed because at the level of methods and fields, there is no mechanism intrinsic to Java with which one can attach additional information like one can attach empty marker interfaces to classes or interfaces. Whether a method, field, local variable, or parameter has been tagged with a tag `@tagname` can be queried using the method `hasTag("tagname")`, and the contents of a certain tag may be retrieved by calling `getTagValue("tagname")`, which returns a `String`.

In this section, we use tags for annotating methods with *preconditions* which are interpreted by CoffeeStrainer constraints. Preconditions (together with postconditions, class invariants, loop variants and loop invariants) have been introduced with the programming language Eiffel [Meyer 1992], which supports *programming by contract*: For each method, the programmer specifies a precondition and a postcondition, which are constraints on the value of parameters, on result values, and on the state of the object that is called. Each method call, then, is subject to the following contract: the caller is required to only issue calls for which the precondition is satisfied at method entry, upon which the callee in turn ensures that the postcondition is satisfied at method exit. Preconditions and postconditions are dynamic constraints that have been proven very useful for producing correct and robust object-oriented programs [Meyer, Nerson 1993]. They are essentially dynamic constraints, i.e. boolean expressions that are checked at run-time whenever a method is called or when it returns.

We do not address method postconditions. In Eiffel, method postconditions may refer to values computed at method entry using the `old` construct. Supporting `old` requires splitting the constraint expression into parts which are evaluated at method entry which then can be used to compute the overall result at method exit. A number of techniques for adding method pre- and postconditions to Java have been proposed [Kramer 1998][Duncan, Hoelzle 1998][Fischer, Meemken 1998]. Although all of these proposals could be added to CoffeeStrainer from the perspective of the implementor of CoffeeStrainer, we believe that it is worth investigating whether there is a set of primitives which would allow a constraint programmer not only to implement method postconditions, but also to implement other kinds of advanced dynamic constraints. Thus, rather than including ad-hoc primitives just for supporting the implementation of method postconditions, we have chosen to leave this as an issue of further research.

In Chapter 2, method preconditions were classified as method usage constraints, because it is the usage (call) of a method which possibly causes constraint violations. From an implementation point of view, it is easier to implement preconditions as dynamic method definition constraints because then the constraint expression may refer directly to the variables accessible at method entry, i.e. to arguments and to the current object's fields. Consider the following example, where the `Object` parameter of method `push` is required not to be `null`:

```
/**
 *@constraints
 * public void checkConcreteMethod(ConcreteMethod m) {
 *   return preRuntime(m.getBody(), "o!=null");
 * }
 */
public void push(Object o) {
  // code for push...
}
```

This example shows that method preconditions can be mapped to dynamic method definition constraints in a straightforward way, but that there is considerable syntactic overhead involved. This overhead can be reduced significantly using tags. The idea is to introduce a tag "`@pre`" which contains the precondition as a boolean expression. An empty marker interface, `ProgrammingByContract`, shown in Figure 3.13, contains a method definition

constraint which interprets "@pre" tags for all methods in subtypes of ProgrammingByContract. Note how the constraint implementation makes use of the non-strictness of the disjunction operator "||" in Java: If there is no @pre tag, preRuntime will not be evaluated and thus no dynamic check will be inserted.

For example, the precondition for push can now be written as follows:

```
/**@pre o!=null*/ public void push(Object o) {
  // code for push...
}
```

```
/**
 *@constraints
 * public void checkConcreteMethod(ConcreteMethod m) {
 *   return !m.hasTag("pre")
 *          || preRuntime(m.getBody(),
 *                        m.getTag("pre"));
 * }
 */
public interface ProgrammingByContract {
}
```

Figure 3.13: Simple implementation of preconditions

The simple implementation of Figure 3.13 does not take inheritance into account. Since preconditions may be weakened in subclasses, checking preconditions in the presence of inheritance requires checking the disjunction of all preconditions specified in the called method and all the methods it overrides. Figure 3.14 shows an updated version of ProgrammingByContract.

The helper method getPrecondition(m) returns, as a string, the disjunction of all preconditions specified in m and all methods that m overrides. Because the empty string is no valid boolean expression, the run-time check is only inserted if getPrecondition returned a non-empty string.

Note that programmers who want to use preconditions do not need to understand (or reproduce) the implementation presented in this section. Rather, they may enable checking of preconditions by marking their classes with the pre-defined empty marker interface ProgrammingByContract.

```
/**
 *@constraints
 * private String getPrecondition(AMethod m) {
 *   StringBuffer result = new StringBuffer();
 *   if(m.getOverriddenMethod()!=null) {
 *     String inheritedPre = getPrecondition(m.getOverriddenMethod());
 *     if(!inheritedPre.equals(""))
 *       result.append("(" + inheritedPre + ")");
 *   }
 *   if(m.hasTag("pre") {
 *     if(!result.toString().equals(""))
 *       result.append(" || ");
 *     result.append("("+m.getTag("pre")+")");
 *   return result.toString();
 * }
 * public void checkConcreteMethod(ConcreteMethod m) {
 *   String precondition = getPrecondition(m);
 *   return precondition.equals("")
 *          || preRuntime(m.getBody(), precondition);
 * }
 */
public interface ProgrammingByContract {
}
```

Figure 3.14: Complete implementation of preconditions