4 CoffeeStrainer Virtues and Limitations

In this chapter, we explain CoffeeStrainer's virtues and limitations and the design decisions that led to them.

To illustrate the points we want to make, we contrast CoffeeStrainer with a hypothetical, simple-minded system for checking constraints on Java programs, called *SimConC* (Simple Constraint Checker). In SimConC, a constraint is written as a function `boolean check(ASG a)` that returns `true` iff the argument ASG `a` represents a program that satisfies the constraint. The SimConC ASG consists of class and interface declarations and method signatures which can be retrieved easily from byte-code files (class files). Constraints are specified in constraint implementation files separate from the source code of the program they apply to. The constraint language is a special-purpose side-effect free language that includes features for pattern matching and tree traversal. Each constraint is applied to the ASG representing the whole program; the required traversal of the ASG has to be provided by the constraint programmer.

Obviously, CoffeeStrainer looks favorable when compared with this system. However, the comparison allows to explain the design choices for CoffeeStrainer in a more concrete way. For a real comparison of CoffeeStrainer with other systems for checking programmer-defined constraints, see Chapter 7.

In the three sections of this chapter, we explain CoffeeStrainer's virtues and limitation in the three aspects of *comprehensiveness* (Section 4.1), *pragmatism* (Section 4.2), and *sophistication* (Section 4.3).

4.1 Comprehensiveness

We believe that it is important to apply a system for checking constraints to real-world examples using a real-world language to test its suitability. For this, it is important that the system is fully implemented. Moreover, the system needs to provide access to the complete abstract semantics graph of the program that should be checked (Section 4.1.1), and it should be based on parsing source code (Section 4.1.2). Limitations with regard to comprehensiveness are discussed in Section 4.1.3.

4.1.1 Complete abstract semantics graph

In SimConC, the ASG does not contain method bodies, as the designer of SimConC believes that method bodies are not important for constraints. After all, the constraints are meant

for users of a class, or for implementors of subclasses — people who are concerned with the interfaces supported by a class, not by their implementation. He assumes that they would not read the method bodies.

In contrast, CoffeeStrainer contains a complete compiler front-end for Java. Method bodies are represented in the ASG, all statements and expressions that make up a method's implementation are accessible. Furthermore, the ASG contains semantic information about methods called by a method call expression, types referred to by a variable declaration, etc. Finally, it is possible to traverse the tree part of the ASG both from containing objects to contained objects and vice versa.

It is easy to see that many of the constraints that have been described in Chapter 2 which can be implemented using CoffeeStrainer could not be implemented with SimConC. For example, the method definition constraint in class `java.awt.Container` (see Section 2.2.2) that required overriding methods to call the overridden method as the first statement obviously needs to look at method bodies. Furthermore, for implementing the constraint, it is important to check that the method called by the first statement is indeed the overridden method. Finally, the method usage constraint in class `java.awt.Component` (see Section 2.2.4) that restricted accessing a certain method to specific packages needs to traverse the tree part of the ASG from contained objects to containing objects to find out the containing package of certain method call expressions.

4.1.2 Based on source code

The designer of SimConC decided to avoid writing a parser for Java source code. Instead, SimConC reads byte-code files (class files), which are easy to parse but still contain most of the information of Java source code, as demonstrated by the existence of decompilers for Java [Proebsting, Watterson 1997]. (Taking into account that SimConC only provides class and method signatures in its ASG, parsing could even be avoided altogether by using Java's reflection facilities.)

In contrast, CoffeeStrainer parses Java source code. The tool is made for programmers, who think about Java in terms of source code, not byte-code instructions. Therefore, the constraint methods refer to syntactic elements of the Java language rather than to specific patterns of byte codes. Moreover, parsing source code makes it possible to include original source code snippets and line numbers in the warning messages that are provided when constraints are violated. Thus, checking constraints at compile-time is appropriate.

4.1.3 Limitations

CoffeeStrainer has three limitations regarding comprehensiveness:

- Of class files (byte-code files), only the class declarations and method signatures are parsed. Thus, the bodies of methods which only exist in byte-code form are not accessible for constraint checking. This limitation could be overcome in two ways: First, one could make the method bodies accessible *as is*, i.e., as byte-code instructions. Although this would be easy from an implementation point of view, it would mean that

a constraint programmer has to understand the byte-code instruction language, and to implement most of the constraints in a variant which is checked on source code and another variant which is checked on byte-code. Therefore, we prefer the second way of fully supporting byte-code files, namely, to use a decompiler for regenerating a source code representation[1]. As has been shown [Proebsting, Watterson 1997], this is possible for all byte-code files. Unfortunately, there is no freely available decompiler for Java.

- Sometimes, constraints need to be re-checked at load-time. For example, this is the case if constraints specify security properties of the software and it cannot be guaranteed at link-time that the software has indeed been checked by CoffeeStrainer. Currently, CoffeeStrainer lacks a mechanism which could ensure this. Moreover, it is not just a technical problem. For example, warning messages would have to be different because they are addressed to the user of the software rather than to the programmer, because in Java, class loading (the equivalent of what is normally linking) is performed just before run-time by the user.

- The third limitation regards inner classes, which are probably not sufficiently supported. This is because inner classes are a relatively new feature of the language Java, and examples for constraints regarding them are still lacking. For example, the constraints found in the Java standard classes do not address inner classes.

4.2 Pragmatic choices

When in doubt, we have taken design decisions that lead to a system that is more useable for everyday programmers. Sections 4.2.1 to 4.2.7 discuss these design decisions, and Section 4.2.8 addresses the limitations of CoffeeStrainer resulting from them.

4.2.1 Integration with base-level code

In SimConC, constraints are defined in separate files, leading to a clear separation between the constraints and the base-level program. The designer of SimConC believes that lead programmers are responsible for writing constraints, and that everyday programmers would not touch the implementation of constraints — if a constraint is violated, it is the error message rather than the constraint implementation which is interesting to a programmer who caused the violation.

In contrast, CoffeeStrainer constraints are embedded within the source code of the program elements they are associated with. The constraint code is extracted from Javadoc comments instead of having separate files that specify constraints that should be applied to certain parts of a program. Thus, the base program and the constraints that apply to it are closely

[1] Note that two different Java source code files could result in the same compiled byte codes - the compilation process is not lossless. For example, the names of local variables might not be available at the byte code level. Thus, checking constraint on source code which was re-generated by a decompiler is not perfect - under certain circumstances, it might produce different results than would be obtained by checking the constraints on the original source code.

tied together. We consider the constraints to be part of the implementation — similar to static types, which in principle could also be specified in separate files, but are so closely tied to the program elements they apply to that it is best to see both at the same time. Integrating constraints with the program elements with which they are associated allows a programmer to find all constraints that apply to a class he writes by examining its supertypes and the classes, interfaces, methods, and fields of other classes he is using. Most likely, he needs to look at the documentation for these program elements anyway to understand the functionality he is using or extending, so it is a natural place to let the constraints be part of the documentation.

4.2.2 No change to base-level language

For SimConC, changing the base-level language is a non-issue since constraints are specified in separate files.

For CoffeeStrainer, it is important to consider how to integrate constraints into base-level code without having to change the base-level language. Fortunately, there is a simple solution: The constraint code is embedded in Javadoc comments just before the associated program elements, leaving syntax and semantics of Java programs unchanged. Thus, it is still possible to use whatever compiler, integrated development environment, or other tools the programmer prefers. Even integrated development evironments which are not file-based but store source code in their own repository retain Javadoc comments, and therefore also retain CoffeeStrainer constraints.

4.2.3 Java as the constraint language

Constraints essentially are expressions of type `boolean` operating on ASG nodes. The designer of SimConC decided to define a special purpose constraint language. This language, which includes constructs for pattern matching and traversing trees makes it easy to program traversals of the ASG. Furthermore, it includes additional boolean operators like "forall".

In contrast, CoffeeStrainer avoids the language design trap [Hoare 1987] as much as possible by using a framework-like structure. Constraints in CoffeeStrainer are Java expressions of type `boolean`; syntax and semantics of constraint methods are just the same as in the Java language. The only "language design" embodied in CoffeeStrainer is its framework structure with the prescribed names for constraint methods and conventions like the variable `rationale`. Although adopting a special-purpose language could allow constraints to be specified in a more concise way, we do not consider inventing a new language a realistic option because using CoffeeStrainer should require as little effort as possible from programmers who already know Java. The requirement to learn a new language would turn away many potential users.

An important advantage of using Java as the constraint language as opposed to a special-purpose language is that the strong static typing of Java applies to the constraint implementations. Defining and implementing a static type system for a special-purpose language is complex and non-trivial, so that in practice special-purpose languages very often lack a static type system.

As shown in the next section, the use of the Visitor design pattern leads to an acceptable structure for constraints using only features of Java that are currently available. Thus, in principle, we consider the issue of language design orthogonal to the issue of providing a convenient way of traversing and accessing the ASG of a program. Advanced language features like pattern matching certainly make sense not only for a constraint language, but for programming languages like Java in general and should therefore be an issue of programming language research in general.

The constraint classes generated by CoffeeStrainer are not hidden from the constraint programmer. Instead, the generated classes form a structure parallel to the original classes, making it possible to refer to meta-level classes by name and reusing constraint methods of other generated constraint classes. Additionally, this allows to write constraint classes that are not generated from source code, which is interesting for two reasons: First, constraints can be specified for classes whose source code is not available (these constraints are then checked on the source code of classes that use or extend them), and second, generated constraint classes can be distributed without giving away the source code of the class from which they are generated.

4.2.4 Predefined tree traversal

In SimConC, it is the constraint programmer's task to write code for traversing the ASG.

In CoffeeStrainer, the traversal of the ASG is implicit — constraint methods are visitor methods that are called during the traversal that is performed once by CoffeeStrainer. Usually, constraint methods do not contain code for traversing the ASG; instead, they query certain properties of a very local part of the ASG. This makes CoffeeStrainer a good fit for constraints that can be expressed by a conjunction of locally-applicable constraint methods. In fact, most programmer-defined constraints are of this type; Chapter 2 presents evidence for this.

4.2.5 Separate checking of compilation units

In SimConC, the ASG of a whole program needs to be provided for the methods that implement constraints, enabling constraints based on whole program analysis.

In contrast, CoffeeStrainer — like compilers for Java — operates on one compilation unit at a time, loading other files only if needed for the analysis or checking of the current compilation unit. This corresponds to the Java philosophy of separating the different compilation units for as long as possible — until load-time. Thus, CoffeeStrainer does not support global analyses very well. For non-iterative analyses, tags and attributes help structuring the traversal so that each program element needs only traversed once. Iterative analyses that compute a fix point, although possible due to CoffeeStrainer's openness, are not a good match for CoffeeStrainer. However, constraints that would require whole-program analysis are not very common.

4.2.6 Efficiency

Efficiency is a problem for SimConC, because the traversal needed for each constraint would be hard-coded into the constraint implementation, leading, in the worst case, to one com-

plete traversal of the ASG for each constraint. Usually, a single traversal would be sufficient; this traversal could be shared by all constraint implementations would be sufficient. A special-purpose language could help if it allowed factoring out the traversal code from each constraint.

In CoffeeStrainer, constraint implementations usually do not contain traversal code. The traversal is performed only once by CoffeeStrainer, and visitor methods are called during that traversal. Experience shows that constraint implementations usually check local properties only, so that the constraint programmer does not have to write traversal code, avoiding duplicate traversals.

Note that the single traversal of the ASGs of compilation units together with separate checking naturally leads to a performance that is similar to a compiler's — see Section 6.3 for a performance comparison of CoffeeStrainer and Sun's Java compiler JAVAC. We believe that it is acceptable for software developers to use tools whose running time is comparable to that of a compiler. Thus, we did not try to exploit obvious optimization opportunities. For example, one such opportunity which has not been realized yet is the observation that not all parts of a program's ASG need to be visited to check the constraints.

4.2.7 Openness

The special-purpose constraint language of SimConC is a declarative language in which only side-effect-free computations can be expressed.

Rather than restricting constraints to be side-effect-free boolean expressions, CoffeeStrainer allows arbitrary Java code in constraint methods, although this makes it possible to write constraints that make use of imperative code instead of (declaratively) returning a value. This decision makes CoffeeStrainer an open, flexible system, enabling uses that its designer did not foresee. However, in the normal case constraints should be specified as declaratively as possible, making as little use of imperative features as possible.

4.2.8 Limitations

Besides the obvious limitations due to the use of Java, a non-declarative language, for specifying constraints, the predefined ASG traversal of CoffeeStrainer together with the separate checking of compilation units makes CoffeeStrainer not particularly well-suited to global analyses. However, this is not a strict limitation — see Section 6.1.4 on the Visitor design pattern and Section 6.1.5 on lazily-evaluated attributes for techniques which help implementing global analyses in a modular and efficient way. It should also be noted that the examples of constraints we have found in the Java standard classes do not require global analyses.

4.3 Elegance

In this section, we describe CoffeeStrainer virtues which are not apparent at first sight. Section 4.3.1 discusses in which ways CoffeeStrainer constraints can be considered modular and

how this enables extension, refinement, and reuse of constraints. In Section 4.3.2, we explain why special support for usage constraints is helpful, using a non-trivial example. Section 4.3.3 shows what can be accomplished by combining static and dynamic constraints, and finally, Section 4.3.4 discusses limitations with regard to modularity, usage constraints, and the combination of dynamic and static constraints.

4.3.1 Modularity

SimConC's constraints are monolithic entities. From the perspective of a programmer whose program is to be checked against a number of constraints, it is difficult to find those constraints that apply to his particular code, because each constraint is applied globally. From the perspective of a constraint designer, there is no easy way to reuse existing constraints, or to extend and refine constraints incrementally.

In contrast, CoffeeStrainer constraints are modular and have an internal structure that helps extending and refining them in subtypes. The scope of a CoffeeStrainer constraint is determined by the program element associated with the constraint and the names of the implemented check methods, and not by its implementation.

It is possible to reuse constraint methods for implementing new constraints. The easiest way of reusing constraint code is by subtyping: Constraints that are associated with a type apply to all subtypes of that type as well. The most flexibility can be gained by using empty interfaces that have associated constraints, in which case constraints are composable by multiply extending from a number of interfaces.

In Java, it is a common technique to use empty interfaces for marking classes with certain properties[2]. This technique works for CoffeeStrainer constraints as well: Both examples of type constraints, `AllFieldsArePrivate` of Section 3.1.2, and `IdentityComparisonDisallowed` of Section 4.3.2 were empty interfaces that can be used to mark classes for which the constraints associated with the empty interface should be checked.

In this way, it is possible to form composed constraint sets by defining empty interfaces that extend several other empty interfaces containing constraints. For example, assume the existence of the following empty interfaces containing constraints: `AllFieldsArePrivate` (all declared fields should be declared private, see Section 3.1.2), `ProvidesAccessorMethods` (for each field, there should be accessor methods without additional side-effects), and `CreationWithFactoryMethods` (objects should be created using factory methods, not using constructors directly). Using these empty interfaces, it is now possible to define

```
public interface MyCodingConventions
  extends AllFieldsArePrivate,
          ProvidesAccessorMethods,
          CreationWithFactoryMethods
{
}
```

[2]Examples of such empty interfaces in the Java standard classes include `java.io.Serializable`, used for marking classes whose objects may be stored in and retrieved from streams, and `java.lang.Cloneable`, used for marking classes whose objects may be cloned using the method `java.lang.Object.clone()`.

Using the empty interface `MyCodingConventions`, it is now possible to mark all classes that should adhere to the coding conventions and have the conjunction of all constraints from the three previously unrelated constraints checked on all such classes. It can easily be seen how this enables the creation of reusable constraint libraries which can be used by programmers who do not want to implement constraints themselves.

4.3.2 Usage constraints

Many examples of constraints that we have found are based on the *usage* of a type, a method, or a field. In SimConC, these can be implemented only by traversing the full ASG for each constraint to find all places in the program where the program element in question is used.

It is instructive to compare this with a situation where only definition constraint methods were available in CoffeeStrainer. It would be awkward to express usage constraints then, because the only scope that includes all possible uses of a type etc. would be the whole ASG. Therefore, expressing usage constraints with definition constraint methods would require all such constraints to be associated with `java.lang.Object`, the root of the inheritance hierarchy.

While usages of methods or fields are only method calls or field accesses, the situation is more complicated for classes and interfaces. There are two ways in which a class or interface can be used in a Java program: First, the name of the class or interface can be referenced directly in, e.g., declarations, object allocations, or cast expressions, etc. Second, the class or interface can be the static type of a certain expression. In CoffeeStrainer, usage constraint methods that correspond to the latter kind of using types are not covered by a single usage constraint method `checkUseAtExpression`. Rather, each use of a type by means of the static type of an expression can be constrained from the viewpoint of its context, as, e.g., in `checkUseAtAssignmentLValue` — check an expression of a certain type which is the l-value of an assignment. Differentiating between the respective contexts of expressions is important, as the alternative of having just `checkUseAtExpression` would mean that the distinction between different contexts would have to be inside virtually all usage constraint methods.

The advantage of having support for usage constraints, and of providing the context for usages of types in expressions can best be demonstrated by means of a non-trivial example:

In object-oriented languages, objects have an identity which does not change over the object's lifetime, whereas the objects' states may change. Accordingly, one can distinguish between *object identity* (checking whether two object references refer to the same object using "==") and *object equality* (checking whether the objects referred to by `o1` and `o2` are equal using `o1.equals(o2)`), the latter of which is usually implemented differently for each class, usually based on the current object's state in comparison with the other object's state.

Often, if object equality for objects of some class is defined (by implementing `equals()`), object identity should not be used by clients of this class. However, inexperienced programmers sometimes are not aware of the difference between object identity and object equality, and use object identity even for objects of classes that should only be compared using object equality. One example of such a class is `java.lang.String`. It is possible that two object

references refer to two different string objects that therefore are not identical; but these two objects may be equal because they contain the same character sequence.

To implement this constraint, we can define an empty interface `IdentityComparisonDisallowed` which captures the constraint that objects whose classes implement this interface may not be compared using object identity. We do allow, however, comparing object references against the object reference `null`.

```
/**
 *@constraints
 * protected boolean isNull(AExpression e) {
 *   if(e instanceof Literal) {
 *     Literal l = (Literal)e;
 *     if(l.constantValue() == null) return true;
 *   }
 *   return false;
 * }
 * public boolean checkUseAtBinaryOperation(
 *                              BinaryOperation bo){
 *   rationale = "objects of this type may not  be compared using == or !=";
 *   return isNull(bo.getLeftOperand())
 *          || isNull(bo.getRightOperand());
 * }
 */
public interface IdentityComparisonDisallowed {
}
```

In this example, we are concerned with the correct use of a type. For this purpose, we have implemented the usage constraint method `checkUseAtBinaryOperation`, which is called whenever an object reference of type `IdentityComparisonDisallowed` is used in the context of a comparison (i.e., in a `BinaryOperation` where the operator is "`==`" or "`!=`"; other binary operations are not defined for object types). Note that this usage constraint method is called whenever `IdentityComparisonDisallowed` or one of its subtypes is used in a reference comparison, and thus may apply globally if every class that is checked by CoffeeStrainer uses `NoIdentity` or one of its subtypes.

The implementation of `checkUseAtBinaryOperation` reflects the constraint that using an object whose class implements `IdentityComparisonDisallowed` is allowed only if the left operand or the right operand of the comparison is the base-level literal `null`. The helper method `isNull(e)` (line 1) returns `true` if the expression `e` is the literal `null`.

Note that it would be not as easy to implement this constraint if the context of a certain usage of a type would not be reflected, i.e., if, in our example, it was not possible to just implement `checkUseAtBinaryOperation` but instead needed to implement `checkUseAtExpression`, requiring to examine the context of the expression and applying the check only if the constraint was a `BinaryOperation`.

4.3.3 Combining static and dynamic constraints

This section describes how checking dynamic constraints enables the constraint programmer to make a conservative constraint more accurate. This will be demonstrated by means of a

non-trivial example in which we would like to make sure that values of a certain object type may never be the object reference `null`, i.e., that the application using values of such a type need not check for `null`.

Because object types in Java are reference types (as opposed to value types like `int`, `float`, etc.), the value `null` is a valid value for all fields, variables, parameters, and method results of reference types. Sometimes, the value `null` is not used for certain types (as, for example, the "Null Object" design pattern [Woolf 1998] suggests), in which case it might be useful to define a constraint makes no sense for a certain type, it is useful to . In an empty interface `NullValueInvalid` with which enumeration classes can be marked, we can define a constraint that specifies that only non-null values should be used for initializing or assigning to fields and variables, for binding to parameters, and for returning from methods. Note that the following are only static constraints:

```
/**
 *@constraints
 * protected boolean isNull(AExpression e) {
 *   return e==null;
 * }
 * protected boolean isNullLiteral(AExpression e) {
 *   return (e instanceof Literal)
 *          && ((Literal)e).constantValue()==null;
 * }
 * protected boolean isNonNull(AExpression e) {
 *   return !isNull(e) && !isNullLiteral(e);
 * }
 * public boolean checkUseAtField(Field f) {
 *   rationale = "field needs non-null initializer";
 *   return isNonNull(f.getInitializer());
 * }
 * public boolean checkUseAtLocalVariable(LocalVariable v){
 *   rationale = "variable needs non-null initializer";
 *   return isNonNull(v.getInitializer());
 * }
 * public boolean checkUseAtReturn(Return r) {
 *   rationale = "method may not return null";
 *   return isNonNull(r.getExpression());
 * }
 * public boolean checkUseAtAssignment(Assignment a) {
 *   rationale = "assignment may not assign null";
 *   return isNonNull(a.getOperand());
 * }
 * public boolean checkUseAtMethodCallParameter(
 *                           int index, AMethodCall mc) {
 *   rationale = "method call argument may not be null";
 *   return isNonNull(mc.getArguments().get(index));
 * }
 * public boolean checkUseAtCast(Cast c) {
 *   rationale = "downcast is not allowed";
 *   return false;
 * }
 * public boolean checkUseAtConditionalIfTrue(
```

```
 *                                          Conditional c) {
 *   rationale = "hiding null in conditional expression" +
 *               " is not allowed";
 *   return isNonNull(c.getIfTrue());
 * }
 * public boolean checkUseAtConditionalIfFalse(
 *                                          Conditional c) {
 *   rationale = "hiding null in conditional expression" +
 *               " is not allowed";
 *   return isNonNull(c.getIfFalse());
 * }
 */
public interface NullValueInvalid {
}
```

Note that although the example might seem long, it is complete and deals with Java and not a toy language. We have defined three helper methods: `isNull` returns `true` if the argument expression does not exist (i.e., is `null`). The method `isNullLiteral` returns `true` if its argument is a `Literal` object representing the constant `null`. The method `isNonNull` returns `true` if its argument expression `e` exists, and is not the base-level literal `null`.

The remaining usage constraint methods check that such expressions are never used for initializing fields or variables, for returning from a method, as the right hand side of an assignment, or passed as argument of a method call, respectively. As a result, code that deals with classes that implement `NullValueInvalid` never has to check for `null` values.

Unfortunately, programming with `NullValueInvalid` is difficult, because downcasts are not allowed (see `checkUseAtCast`). This makes it impossible to store and later retrieve such objects from the standard Java collection classes, because retrieving objects from collections usually involves a downcast.

We will now change the implementation of `checkUseAtCast` to a dynamic constraint that makes the check less conservative and more accurate, solving the problem that one would like to allow storing `NullValueInvalid` objects in collections:

```
 * public boolean checkUseAtCast(Cast c) {
 *   rationale = "downcast of null value is not allowed";
 *   return postRuntime(c.getExpression(), "$value!=null");
 * }
```

After changing the usage constraint method `checkUseAtCast`, usages of casts for types derived from `NullValueInvalid` will no longer be statically flagged as constraint violations by CoffeeStrainer. Rather, after each evaluation of an expression of type `NullValueInvalid` that is about to be casted, a run-time check will be inserted which makes sure that the value for the cast is not the value `null`. Note that in Java, it is legal to downcast the value `null`. Thus, the dynamic check really is needed to ensure that in the presence of casts, variables of type `NullValueInvalid` can never contain the value `null`.

4.3.4 Limitations

Again, we are aware of a number of limitations due to the design decisions on which CoffeeStrainer is based:

- Although empty marker interfaces are an elegant technique for reusing constraint implementations, this works only on the level of classes and interfaces. CoffeeStrainer does not support similar techniques on the level of methods and fields. However, it is possible to factor out common code from method and field constraints as a programming discipline due to CoffeeStrainer's openness.
- The context information provided for usage constraints that deal with expressions may not be sufficient for all purposes. The example constraint in Section 4.3.2, associated with the empty marker interface `IdentityComparisonDisallowed`, made good use of the one-level context information provided by CoffeeStrainer by implementing the usage constraint method `checkUseAtBinaryOperation`. In some cases, more than one level of context information might be useful. For example, one might want to implement a hypothetical usage constraint method "`checkUseAtBinaryOperationInIf`" to disallow object reference comparisons only in `if` statements. However, we did not find enough examples of such constraints which would justify the additional complexity of providing more than one level of context information.
- The side-effects caused by using `atRuntime` for implementing dynamic constraints needs special attention: For example, it is important to understand that given two strings `x` and `y` containing run-time checks, there is a difference between the CoffeeStrainer constraint "`atRuntime(x) || atRuntime(y)`" and "`atRuntime(x + " || " + y)`". Because `atRuntime` always returns `true`, the check contained in `y` will not be inserted in the first case because the boolean operator "`||`" is non-strict in Java.