COFFEESTRAINER - Statically Checking Structural Constraints on Java Programs

Boris Bokowski

bokowski@inf.fu-berlin.de

Technical Report B-98-14
December 1998

Abstract

It is generally desirable to detect program errors as early as possible during software development. Statically typed languages allow many errors to be detected at compile-time. However, many errors that could be detected statically cannot be expressed using today's type systems. In this paper, we describe a meta-programming framework for Java which allows for static checking of structural constraints. In particular, we address how design principles and coding rules can be captured.

Keywords: reflection and metaobject models, frameworks, software engineering practices

Freie Universität Berlin
Institut für Informatik
Takustrasse 9
D-14195 Berlin, Germany

1. Introduction

It is generally desirable to detect program errors as early as possible during software development. Statically typed languages allow many errors to be detected at compile-time. However, many problems that could be detected statically cannot be expressed using today's type systems. In fact, in any reasonably sized software development project, rules constraining the structure of the application under development must be obeyed by the programmers, ranging from simple coding conventions to design constraints caused by, e.g., using design patterns. Although most of these constraints could be enforced at compile-time, there exists little support for statically checking programmer-defined constraints which is both expressive and useable for everyday programmers.

This paper presents a system called CoffeeStrainer[1] that supports compile-time checking of programmer-defined constraints concerning a program's structure. The system is useful for several reasons: for *software development teams,* the system allows to specify coding rules that will be enforced statically; for *framework developers,* it allows to specify rules for using the framework correctly; for *framework users,* it can warn of incorrect uses or specializations of a framework.

Compared with previous work on specifying implementation or design constraints for object-oriented programs [Chowdhury, Meyers 93; Minsky 96; Klarlund et al. 96], CoffeeStrainer is different in the following aspects:

- Instead of defining a new special-purpose language, constraints can be specified in Java, a language the programmer already knows;
- The system is implemented as an open object-oriented framework for compile-time meta programming that can be extended and modified by defining new object-oriented meta-level abstractions;
- The meta-level code and the base-level code share the same structure, making it easy to find the rules that apply to a given part of the program;
- The meta-level code is embedded in special comments, leaving the base-level syntax and semantics unchanged; thus, arbitrary compilers and other tools can operate on the source code;
- When defining a new rule, the programmer has access to a meta model that is a complete object-oriented representation of the program that is to be checked; the meta model is not restricted to classes, methods and method calls;
- Special support is provided for constraining the usage of classes and interfaces.

Following [Chowdhury, Meyers 93], there are three categories of constraints: *stylistic constraints* are concerned with names and other aspects of a program that, when changed, do not affect its semantics; *implementation constraints* deal with problematic language constructs or cover common traps and pitfalls that may easily lead to subtle programming errors; and *design constraints* reflect programming rules for the

[1] CoffeeStrainer has been implemented in Java and is available at http://www.inf.fu-berlin.de/~bokowski/CoffeeStrainer

correct use of a framework, or coding conventions resulting from the use of design patterns. In the remainder of this section, we will list quite a large number of examples for such constraints, both because we want to convey the broad scope of constraints that can be specified and checked with CoffeeStrainer, and because we want to show that such constraints are ubiquitous in any software development project.

Examples for stylistic constraints that can be specified with our system and checked at compile-time are:

- Package names should be lowercase only;
- In a class definition, the declarations of public variables, public constructors, and public methods should precede the private declarations;
- The scope of local variables should be minimal, i.e. a variable declaration should be in the smallest block that contains all uses of the variable.

Examples for implementation constraints are:

- A class that provides its own implementation of `public boolean equals(Object other)` should also implement `public int hashCode()` and vice-versa, because equal objects must have the same hash code to be correctly added to and removed from hash-based collections;
- Branches of an if-statement should be blocks rather than single statements, because when adding a new statement to a single-statement branch, programmers often forget to correctly group both statements in a block;
- String objects should be compared using the method `equals` rather than the identity operator "==".

Design constraints can be classified further into *coding conventions* which may be defined for an organization, a single project, or a part of a single program, aimed at code that is easier to comprehend and maintain; *inheritance constraints* that specify a contract between a class and its subclasses, or rules that have to be followed by a class to correctly implement a specific interface, sometimes called inheritance contracts or reuse contracts [Steyaert et al. 96]; and *usage constraints* that constrain the way in which objects of a certain type may be used.

Examples for coding conventions that can be specified using CoffeeStrainer are:

- All instance variables (fields) should have private access only, and accessor methods should be provided that have no other side effects;
- When using Java RMI, classes that implement the interface `java.net.Remote` should not be used in variable or field declarations; instead, interfaces derived from `Remote` should be used, such that remote objects can always be substituted for local objects;
- Classes from vendor- or platform-specific packages, such as sun.* or com.microsoft.*, should not be used because the system under development shall be certified as 100% pure Java.

Inheritance constraints include for example:

- All methods in subclasses of a class `C` overriding a certain method `m()` should call `super.m()` before doing anything else;

- In a certain abstract class, a method `template()` that contains a call "`this.hook()`" should not be changed such that the call to `hook()` is removed, because subclasses rely on the call to `hook()` (see [Pree 95] for a definition of template and hook methods, a design idiom used in many design patterns);

- When using the Visitor pattern [Gamma et al. 95], classes that implement the interface `Visitable` should implement the method `public void accept(Visitor v)` by calling back the visitor object using a type-specific method `public visit<type>(<type> o)`, without doing anything else.

Examples of usage constraints are:

- A specific framework method that for certain reasons needs to have public access (e.g., a public constructor required for object serialization) should not be called by user-level code;

- Fields, variables, parameters, and result values of a certain type (e.g., an enumeration type) may not contain the value `null`, i.e., only non-null values should be used for initializing or assigning to fields and variables, for binding to parameters, and for returning from methods;

- By implementing one of the empty interfaces Layer1, Layer2, Layer3, ..., a class can be marked as belonging to a certain architectural layer; classes should only call methods of classes that belong to the same layer or the layer immediately below.

The remainder of the paper is organized as follows. Section 2, using three example constraints of increasing complexity, explains how constraints are specified in CoffeeStrainer. In section 3, an abstract overview of the meta framework underlying CoffeeStrainer is given. Section 4 compares our proposal with related work. Section 5 draws conclusions and points out directions for future work.

2. Specifying constraints with CoffeeStrainer

In this section, we elaborate on three example design constraints. For each constraint, first, a more detailed description and motivation is given. Second, the constraint is specified using CoffeeStrainer, and it is explained how this constraint specification is used to check the constraint on Java programs.

2.1. Private access for fields

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.

Since this constraint is not appropriate in all cases (for example, in performance-critical applications, the additional indirection might be prohibitive), we define an empty interface called `PrivateFields` and require private access for fields only for classes that implement `PrivateFields`. This interface contains meta-level code that represents the constraint:

```
interface PrivateFields {
    /*/ public void checkField(Field f) {
            if(!f.isPrivate()) {
                reportError(f, "field is not declared private");
            }
        }
    /*/
}
```

Comments that begin and end with "`/*/`" are treated specially in CoffeeStrainer: They enclose code belonging to the meta-level that has access to a complete object-oriented representation of the base-level program.

With CoffeeStrainer, checking a class that implements `PrivateFields` proceeds as follows:

- All classes needed to compile the class will be parsed, and an object structure will be built that represents all classes, interfaces, fields, methods, statements, expressions and so on. This object structure corresponds to an abstract syntax tree enriched by name and type analysis information.

- Eventually, the interface `PrivateFields` will be parsed, and the special comment will be detected. From the code contained in that comment, a new class `meta.PrivateFields` will be generated. This class will then be compiled on-the-fly and loaded dynamically.

- After all necessary files have been parsed, the actual checking will be performed: For each object `f` of type `Field` corresponding to a field declaration contained in a class that implements `PrivateFields`, the method `checkField` will be called by the CoffeeStrainer framework, providing `f` as an argument.

- As can be seen from the code above, for every field object that does not have private access, a method `reportError` (defined in a superclass of all generated classes) will be called that reports the violation of the constraint to the user, including the file and position of the offending construct.

In general, constraints are meta-level code that is embedded into classes and interfaces. This code will be called by the CoffeeStrainer framework. Violations of constraints are reported by calling `reportError()` with appropriate arguments.

2.2. Call method in superclass when overriding

Often, an abstract superclass defines methods that can (and should) be overridden in subclasses. Sometimes, overriding methods are expected to call the overridden method before doing anything else. For instance, consider a class `MediaStream` with a method `initialize()` that performs necessary initializations that could not be performed in the constructor. A subclass of `MediaStream` that requires additional intialization actions should override `initialize()`, call `super.initialize()` as its first action, and then perform subclass-specific initializations. This constraint is captured as follows:

```
abstract class MediaStream {
(2)   public void initialize() {}
(3)   /*/ public void checkConcreteMethod(ConcreteMethod m) {
(4)             if(m.containingClass() == instance) return;
(5)             if(!m.getName().equals("initialize")) return;
(6)             Astatement first;
(7)             first = (AStatement) m.getBody().getStatements().
(8)                                                   firstElement();
(9)             if(first!=null && (first instanceof ExpressionStatement)) {
(10)                 Expression e;
(11)                 e = ((ExpressionStatement) first).getExpression();
(12)                 if(e instanceof InstanceMethodCall) {
(13)                      InstanceMethodCall c;
(14)                      c = (InstanceMethodCall) e;
(15)                      if ((c.getInstance() instanceof Super) &&
(16)                           c.getCalledMethod().getName().equals(
(17)                                "initialize")) {
(18)                           return;
(19)             }    }    }
(20)             reportError(m, "does not call super.initialize() as " +
(21)                            "its first statement");
(22)       }
(23)   /*/
}
```

Again, CoffeeStrainer will generate a class `meta.MediaStream` that contains the meta-level code contained in MediaStream. In this case, the method `checkConcreteMethod` will be called for every non-abstract method contained in `MediaStream` or any of its subclasses. To refer to specific lines of the definition of `MediaStream`, the lines have been numbered.

As the constraint should only be checked on subclasses of `MediaStream`, `checkConcreteMethod` returns if the method to be checked is part of `MediaStream` itself (line 4). The static field `instance` is defined in every generated class and contains the class metaobject that represents the base-level class, in this case, a metaobject representing `MediaStream`.

Similarly, if the method to be checked has a name other than `initialize`, `checkConcreteMethod` returns without any further checks (line 5). The remainder of the method checks whether the first statement is a non-static method call (lines 6-14), and whether that method call calls initialize (lines 16-17) on super (line 15). If this is the case, the method returns; otherwise, an error is reported (line 20-21).

This example shows how to check properties of the metaobject structure. Compared with other approaches for specifying code constraints (e.g., CDL [Klarlund et al. 96]), it may seem that our approach lacks conciseness. However, note that we chose not to invent a special-purpose language, but rather use Java - a language the programmer already knows. Clearly, Java is not as concise as a declarative special-purpose language, but the additional "noise" can easily be filtered out by Java programmers. Furthermore, it is possible to employ all object-oriented structuring mechanisms for factoring out common code, and for making complex constraints more declarative.

2.3. Disallow `null` value for a certain type

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. Sometimes, as for example when defining classes that should be used as enumerations, the value `null` should not be used for fields, variables, etc. of that enumeration type. In an empty interface `Enumeration` 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:

```
interface Enumeration {
(2)   /*/  private boolean isNull(AExpression e) {
(3)             if (e == null) return true;
(4)             if (e instanceof Conditional) {
(5)                  Conditional c = (Conditional) e;
(6)                  return isNull(c.getIfTrue()) ||isNull(c.getIfFalse());
(7)             } else if (e instanceof Constant) {
(8)                  Constant c = (Constant) e;
(9)                  if(c.getValue() == null) {
(10)                      return true;
(11)                 }
(12)            }
(13)            return false;
(14)       }
(15)       public void checkUseAtField(Field f) {
(16)            if (isNull(f.getInitializer()) {
(17)                 reportError(f, "may be initialized with null");
(18)       }    }
(19)       public void checkUseAtVariable(Variable v) {
(20)            if (isNull(v.getInitializer()) {
(21)                 reportError(v, "may be initialized with null");
(22)       }    }
(23)       public void checkUseAtReturn(Return r) {
(24)            if (isNull(r.getResult()) {
(25)                 reportError(r, "may return null");
(26)       }    }
(27)       public void checkUseAtAssignment(Assignment a) {
(28)            if (isNull(a.getOperand()) {
(29)                 reportError(a, "may assign null");
(30)       }    }
(31)       public void checkUseAtMethodCallArgument(AMethodCall mc,
(32)                                                 int arg_index) {
(33)            AExpression e = (AExpression) mc.getArguments().
(34)                                               elementAt(arg_index);
(35)            if(isNull(e)) {
(36)                 reportError(mc, "may pass null as argument " +
(37)                                                         arg_index);
(38)       }    }
(39)  /*/
}
```

In the previous two examples, the checks were always concerned with language constructs that should or should not appear in a class (or interface) and all its descendants. This time, we are concerned with the correct usage of a type. For this purpose, a second class of methods can be defined in meta-level code: methods that start with `checkUseAt` are called by the CoffeeStrainer framework for every use of a class or interface (and for uses of subclasses and subinterfaces). For instance, the method `checkUseAtField(Field f)` defined in the meta-level code of class `C` is called for every field declaration that uses `C` or one of its subclasses as the field type.

In this example, the ability to define new methods for factoring out common code is demonstrated. The method `isNull` (lines 2-14) returns true if a metaobject representing an expression does not exist, is itself the constant value `null` or may produce `null` by hiding the constant inside Java's ternary conditional operator. The remaining 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 `Enumeration` never has to check for `null` values!

3. CoffeeStrainer as a meta-programming framework

A meta model is an object-oriented representation of a program. It consists of objects for each of the elements that make up a base-level program: class objects, package objects, method objects, statement objects, variable declaration objects, expression objects, and so on. The meta model for a program can be derived from the object-oriented abstract syntax tree for the program, enriched by additional information obtained by name analysis (associating each use of a name with its declaration) and type analysis (associating each expression with its static type).

3.1. Metaobject model

In CoffeeStrainer, meta-level code is embedded into the base-level program inside special comment sections. Comments inside class definitions that begin and end with the string "`/*/`" are collected and inserted into meta-level classes that are generated on the fly. Supposing the base-level class is called `pack.MyClass`, the newly created meta-level class is called `meta.pack.MyClass`. This class then contains all declarations that have been collected by parsing the special comments.

Assuming that no base-level package called "lang" exists, the package meta.lang contains all classes that comprise the meta object structure. All generated classes, then, inherit from either `meta.lang.Class`, `meta.lang.Interface`, or `meta.lang.Throwable`, depending on whether they represent a base-level class, a base-level interface, or a base-level exception type (i.e., a base-level class that inherits from `java.lang.Throwable`). Additionally, for each class, a static field instance is generated which at meta-level runtime will contain the appropriate singleton metaobject for the corresponding class. For instance, from the interface `PrivateFields`, which was given earlier, the following meta class would be generated:

```
package meta;

class PrivateFields extends meta.lang.Interface {
     public static meta.PrivateFields instance = null;
     public void checkField(Field f) {
          if(!f.isPrivate()) {
               reportError(f, "field is not declared private");
          }
     }
}
```

After generating all meta-level classes, they are compiled and loaded into the system. Before invoking any of the methods in the meta-level classes, for each of these classes, the static field `instance` is initialized appropriately with a metaobject, and the rest of the metaobject structure is built.

Note that the inheritance structure of the meta-level classes does not reflect the inheritance structure of the base-level classes. The reason for this is that for base-level *interfaces*, meta-level *classes* are generated instead of meta-level interfaces such that they may contain meta-level code as well. Thus, multiple implementation inheritance would be needed in the meta-level if one wanted to reflect the base-level interface inheritance structure. Instead, in CoffeeStrainer, a delegation-based approach is employed to enable specialization of meta-level code, which will be described at the end of section 3.2.

3.2. A framework for statically checking constraints

The meta model as described so far can be used for all sorts of meta-level programming. In this section, we will show how the generic meta model is specialized such that rules about a program's structure can be specified easily.

Checking structural properties of a program involves a traversal of its metaobject structure, and performing checks at various points during the traversal depending on the type of the object at hand. Remember that the metaobject structure was built by enriching an abstract syntax tree. This allows us to traverse the structure in a well-defined way, namely by a depth-first traversal of the abstract syntax tree corresponding to the textual structure of the program.

To determine the checks that are performed at each object reached during the traversal, the framework makes use of the Visitor design pattern [Helm et al. 95]. In our system, the Visitor design pattern is used as follows: Singleton instances of generated classes, called *metaclass objects* because they represent base-level interfaces or classes, are used as visitor objects that visit all objects of the metaobject structure. For this purpose, each metaobject class defines a method accept(CheckVisitor v) that calls back the visitor object with a method specific to the visited object's type. For instance, the implementation of accept(CheckVisitor v) in the metaobject class meta.lang.Field calls back the visitor object with v.checkField(this):

```
public void accept(CheckVisitor v) {
     v.checkField(this);
}
```

The classes meta.lang.Class, meta.lang.Interface, and meta.lang.Throwable contain empty definitions for all check-methods, such that only the methods required for checking a specific constraint have to be implemented. Additionally, they contain empty definitions for appropriate checkUseAt-methods, such as, for example, checkUseAtThrows(AMethod m) in class meta.lang.Throwable.

After all generated classes have been compiled and loaded, and after the metaobject structure has been built, constraint checking for one compilation unit proceeds as follows:

- for each interface or class with name X contained in the compilation unit, the metaclass object for X (the singleton object referenced by the static field meta.X.instance) and the metaclass objects for all its superclasses and implemented interfaces are collected.

- for each node n of type <N_type> in a depth-first traversal of the parse tree of X, the following methods are invoked:

    - check<N_Type>(n) is invoked on the metaclass object representing X itself,

    - check<N_Type>(n) is invoked on all interfaces that X implements directly,

- the preceding two steps are performed analogously for all metaclass objects representing superclasses of `X`.

- appropriate `checkUseAt`-methods are invoked on a metaclass object representing a type `Y` if in node `n`, the type `Y` is used. Again, the same `checkUseAt`-method is called for implemented interfaces and superclasses of `Y`.

Note that, for each node of the syntax tree, `check`-methods not only of the meta-level code of the containing class or interface are called, but also for all types that the containing class or interface is derived from. In effect, all checks that are defined in a type hierarchy have to be performed, such that the conjunction of all defined constraints is checked.

4. Related work

CoffeeStrainer is a meta programming framework for specifying constraints about object-oriented programs that are checked at compile-time. Three systems that are very similar to CoffeeStrainer will be discussed in section 4.1, namely CCEL [Chowdhury, Meyers 93], LGA [Minsky 96], and CDL [Klarlund et al. 96]. In section 4.2, CoffeeStrainer as a meta programming framework will be put into the context of other meta programming systems. Related work that shares the idea of specifying constraints and programming rules above those that can be captured by type systems will be discussed in section 4.3.

4.1. Other systems

The C++ Constraint Expression Language - CCEL - [Chowdhury, Meyers 93] allows to specify statically checked constraints on programs written in C++. Like in CoffeeStrainer, constraint specifications in CCEL have access to the meta structure of a program. This meta structure, however, is restricted to the declaration part of C++, i.e., to class, function and variable declarations. Thus, constraints concerning the definitions of program elements, i.e., class and function implementations and variable initializations, cannot be expressed in CCEL. As has been discussed for the last example constraint, extending a formalism for specifying constraints to constrain the usage of program elements in addition to constraining only their declaration, involves more than supplying a richer metaobject structure, namely, a new set of callbacks called `checkUseAt` in addition to the original `check` had to be defined. CCEL defines a new language for specifying constraints, that, although very similar to C++, needs more than three pages of grammar description [Duby et al. 92]. We think that, although CCEL constraint specifications are more concise than specifications for CoffeeStrainer, our decision to use Java makes it easier for the programmer to define new constraints. As has been pointed out earlier, the additional "noise" caused by using Java instead of a special-purpose language can be easily filtered out by Java programmers. Furthermore, constraints in CCEL are kept separate from the program that is to be checked, making it difficult to connect the program's structure with the appropriate constraints. In CCEL, there are no means to write abstractions that can be reused in different constraint specifications.

Law-Governed Architecture - LGA - [Minsky 96] describes regularities in object-oriented software in general by defining an abstract object model. Constraints are specified on the language-independent object model, using Prolog as the constraint language. A mapping of the abstract object model to Eiffel has been defined and implemented [Minsky, Pal 96]. Using LGA, not only statically checkable constraints on object-oriented programs can be specified. Its scope extends on one side to constraints that can be defined on the software development process, and on the other side to constraints that can only be checked at runtime. In this comparison, we will consider the statically checkable subset of LGA only. Although defining language-independent constraints may seem a desirable property, it rules out a large class of useful constraints. In addition to implementation constraints, a lot of design constraints that make use of language-specific features cannot be specified using LGA. Often, constraints caused by using design patterns are of this type. For CoffeeStrainer, adding abstraction layers to the meta model could enable language-independent constraints be specified, without ruling out constraints on language-specific constructs. Like CoffeeStrainer, and unlike CCEL, constraints on the usage of types (classes) can be expressed in LGA. However, these constraints are restricted to operations defined on objects in the abstract object model - object creation and deletion, reading and writing object properties, and invoking methods on objects. As can be seen by the `Enumeration` example, CoffeeStrainer does not have these restrictions: the assignment of values to fields, variables and parameters of a certain type could be constrained. Because Prolog is used as the constraint language, new abstractions can be defined and reused in several constraints. Again, the programmer has to learn a new language, and the constraints are not integrated with the program that is to be checked.

In the work on Formal Design Constraints [Klarlund et al. 96], a constraint language called Category Description Language - CDL - is used. Unlike CCEL and LGA, CDL specifications work on complete parse trees, and thus, implementation constraints and language-specific constraints can be specified. CDL is based on a theory of logics on parse trees that allows concise constraint specifications. This theory allows to decide automatically whether a set of constraints is consistent, i.e., not self-contradictory, whereas in CoffeeStrainer, CCEL and LGA, a set of constraints could be specified for which no conforming program may exist, or a set of constraints that leads to infinite loops during checking. However, because "pure" CDL is not sufficient for practical constraints, it was extended with externally computed predicates, which may lead to the same inconsistency and undecidability problems of the other approaches. Constraints defined in CDL have names that are used to annotate the program being checked, leading to an integration similar to CoffeeStrainer's. However, to allow CDL constraint names to appear in object-oriented programs, the grammar of the implementation language needs to be extended. CoffeeStrainer uses special comments for this integration, such that neither syntax nor semantics of the implementation language is changed.

4.2. Meta programming

Usually, systems for meta programming (for example, the CLOS metaobject protocol [Kiczales et al. 91], or OpenC++ [Chiba 95]) are meant for changing or extending the semantics of an object-oriented programming language. In contrast, CoffeeStrainer does not support changing the semantics, and it also does not require changing the syntax of the base-level language (as in OpenC++). However, it shares some characteristics with these systems:

Like the CLOS metaobject protocol, CoffeeStrainer is designed as an open object-oriented framework that can be extended and specialized for specific purposes.

Like OpenC++, CoffeeStrainer is a system that works in the compilation phase only, having access to an object-oriented representation of the base-level program. Using some of the ideas of CoffeeStrainer, namely, sharing of structure between base-level and meta-level code, using Java at the meta-level as well, and utilizing special comments for meta-level code, a powerful meta programming system for Java could be derived.

4.3. Specifying constraints

In [Helm et al. 90], contracts as a high-level formalism for specifying mutual usage constraints are proposed. Unfortunately, the formalism is too expressive to be statically checkable. [Steyaert et al. 96] introduce the notion of reuse contracts for constraints regarding the contract between a class and its subclasses, such as the second example of section 2. We consider CoffeeStrainer an ideal platform for implementing reuse contracts for Java. [Gil, Eckel 97] present formal definitions of traits, a notion very similar to structural constraints. Again, CoffeeStrainer would be a good candidate for implementing a system that can enforce traits.

5. Conclusions

We have presented a novel meta-programming framework for Java which allows for static checking of structural constraints on Java programs. Unlike previous work, CoffeeStrained does not define a new special-purpose language. Instead, constraints are specified in Java, the same language as the base-level code, and thus a language the programmer already knows. The system is implemented as an open object-oriented framework for compile-time meta programming that can be extended and modified by defining new object-oriented meta-level abstractions. Meta-level code and base-level code share the same structure by embedding meta-level code in special comments, making it easy to find the rules that apply to a given part of the program, and allowing arbitrary compilers and tools to be applied to the source code that contains constraints. When defining a new rule, the programmer has access to a meta model that is a complete object-oriented representation of the program that is to be checked; unlike other proposals, the meta model is not restricted to classes, methods and method calls, and special support is provided for constraining the usage of classes and interfaces. The framework, which has been fully implemented, is structured by using the Visitor design pattern.

An area of further work is the issue of encoding properties of language constructs. In this paper, we have used empty interfaces as "markers" that allow constraints to be specified applying only to marked classes and interfaces. We foresee that for some constraints, similar markers will be needed for methods, fields, variables, parameters, etc. Although sometimes naming conventions might help (e.g., fields whose names begin with "shared_" should be accessed in synchronized methods only), in general, a new kind of special comments will be needed that can be used to annotate arbitrary language constructs.

References

[Chowdhury, Meyers 93] A. Chowdhury, S. Meyers, *Facilitating Software Maintenance by Automated Detection of Constraint Violations*, IEEE Conference on Software Maintenance, Montréal, Quebec, September 1993

[Chiba 95] S. Chiba, *A Metaobject Protocol for C++*, Proceedings OOPSLA'95, October 1995

[Duby et al. 92] C. K. Duby, S. Meyers, S. P. Reiss, *CCEL: A Metalanguage for C++*, Proceedings of USENIX C++ Conference, Portland, Oregon, August 1992

[Gamma et al. 95] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns - Elements of Reusable Object-Oriented Software*, Addison-Wesley 1995

[Gil, Eckel 97] J. Gil, Y. Eckel, *A Framework for Static Checking of Design Level Traits*, OOPSLA '97 poster presentation, http://www.cs.technion.ac.il/~eckel/traits.ps

[Helm et al. 90] R. Helm, I. M. Holland, D. Gangopadhyay, *Contracts - Specifying Behavioural Compositions in Object-Oriented Systems*, Proceedings of OOPSLA'90, ACM Press, 1990

[Kiczales et al. 91] G. Kiczales, J. des Rivières, D. G. Bobrow, *The Art of the Metaobject Protocol*, MIT Press, 1991

[Klarlund et al. 96] N. Klarlund, J. Koistinen, M. I. Schwartzbach, *Formal Design Constraints*, Proceedings of OOPSLA'96, ACM SIGPLAN Notices, Vol. 31, No. 10, October 1996

[Minsky 96] N. H. Minsky, *Law-Governed Regularities in Object Systems; Part 1: An Abstract Model*, Theory and Practice of Object Systems, Vol. II, No. 4, Wiley 1996

[Minsky, Pal 96] N. H. Minsky, P. pratim Pal, *Law-Governed Regularities in Object Systems; Part 2: A Concrete Implementation*, Theory and Practice of Object Systems, Vol. III, Wiley 1997

[Pree 95] W. Pree, *Design Patterns for Object-Oriented Software Development*, Addison-Wesley 1995

[Steyaert et al. 96] P. Steyaert, C. Lucas, K. Mens, T. D'Hondt, *Reuse Contracts: Managing the Evolution of Reusable Assets*, Proceedings of OOPSLA'96, ACM SIGPLAN Notices, Vol. 31, No. 10, October 1996