A Framework for Implementing Pluggable Type Systems

Chris Andreae and James Noble
Victoria University of Wellington
{chris,kjx}@mcs.vuw.ac.nz

Shane Markstrum and Todd Millstein
University of California, Los Angeles
{smarkstr,todd}@cs.ucla.edu

Abstract

Pluggable types have been proposed to support multiple type systems in the same programming language. We have designed and implemented JAVACOP, a program constraint system for implementing practical pluggable type systems for Java. JAVACOP enforces user-defined typing constraints written in a declarative and expressive rule language. We have validated our design by (re)implementing a range of type systems and program checkers. By using a program constraint system to implement pluggable types, programmers are able to check that their programs will operate correctly in restricted environments, adhere to strict programming rules, avoid null pointer errors or scoped memory exceptions, and meet style guidelines, while programming language researchers can easily experiment with novel type systems.

Categories and Subject Descriptors D.1.5 [*Programming Techniques*]: Object-oriented Programming; D.2.1 [*Software Engineering*]: Requirements/Specifications—Tools

General Terms Design, Languages, Reliability

Keywords JavaCOP, pluggable type systems

1. Introduction

Many researchers have noticed that traditional type systems do not enforce all of the kinds of constraints that may be useful for programmers. A common way to address these limitations is to define an extension to a base type system that introduces new constraints on programs, often in conjunction with new program annotations. For example, a `nonnull` annotation can be used to require a variable to never point to `null` [26], a `confined` annotation can be used to ensure that no references to instances of a class can escape the class's package [49], and various kinds of `readonly` annotations can be used to ensure that an object reference is not modified [7].

Unfortunately, language designers cannot anticipate all of the constraints and conventions that programmers will want to specify, nor can they anticipate all of the practical ways in which such properties can be checked. Further, different rules may be important to enforce in different programs, and it would be too unwieldy to require all programs to obey all rules all of the time. Therefore it is desirable to provide a framework for *user-defined type systems*. Bracha has previously identified the need for such a framework, coining the term *pluggable types* for the resulting system [8].

OOPSLA'06 October 22–26, 2006, Portland, Oregon, USA.

Aside from its clear benefits to programmers, a framework for pluggable type systems is also beneficial to programming language researchers, who can use such a framework to experiment easily with novel type systems.

In this paper, we present the design, implementation, and evaluation of a practical framework for implementing pluggable type systems in Java [3, 32], which we call JAVACOP ("Constraints On Programs"). We have designed a declarative rule language in which programmers may specify their pluggable type systems. User-defined rules work over a rich abstract syntax tree (AST) representation for Java 1.5 programs and can refer to user-defined type information, which is expressed as annotations in Java 1.5's metadata facility [6]. JAVACOP then automatically enforces these rules on programs as they are compiled.

As a simple example, consider the following code which uses a `@NonNull` annotation to specify the additional type constraint that the field `firstname` (of Java type `String`) may never be null.

```
class Person {
   @NonNull String firstname = "Chris";
   void setFirstName(String newname)
     { firstname = newname; }
}
```

Clearly, the assignment to `firstname` within `setFirstName` is potentially erroneous, since `newname` may be null. It is straightforward for a user to write a JAVACOP rule that would discover this potential error. For example, the JAVACOP rule below requires the right-hand-side expression of each assignment statement to be demonstrably nonnull whenever the type of the left-hand-side variable or field is declared as such.

```
rule checkNonNull(Assign a){
 where(requiresNonNull(a.lhs)){
  require(definitelyNotNull(a.rhs)):
   error(a,"Possible null assignment"
         + " to @NonNull"); } }
```

This rule relies on two user-defined helper predicates. The `requiresNonNull` predicate checks that the given variable or field was declared with the `@NonNull` attribute. The `definitelyNotNull` predicate inspects the given expression to determine if it is definitely nonnull. For example, `definitelyNotNull` would define literals other than `null` to be nonnull. Assuming `definitelyNotNull` does not allow an arbitrary variable to be considered nonnull, the assignment to `firstname` in `setFirstName` will fail our rule, causing the compiler to output the following error message:

```
Person.java:4: Possible null assignment to
@NonNull
    { firstname = newname; }
      ^
1 error
```

As detailed in Section 5, there have been several prior systems that allow user-defined constraints to be enforced on programs. JAVACOP is distinguished by a novel combination of features that is targeted to support the development of practical pluggable type systems:

- JAVACOP leverages Java 1.5's metadata facility to allow users to introduce additional type specifications. Java's annotations cannot be placed directly on types, but only on declarations. As such, JAVACOP supports two main kinds of annotations for pluggable types. First, an annotation on a class declaration can be used to indicate that all values of the Java type corresponding to that class satisfy some additional properties. For example, the declaration of a class `C` can be annotated as `@Confined` to indicate that no instance of that class will escape the class's package. Second, an annotation on a variable (local or formal) or field declaration can be used to indicate that a particular variable or field satisfies some additional properties. For example, in our `Person` class above, the `firstname` field is declared to be both a `String` and to be never null.
- JAVACOP employs a declarative, rule-based language for expressing the semantics of these new annotations. While it would be possible to write the rules directly in Java with respect to an AST API, we believe rules in JAVACOP's language will be significantly easier for rule designers and programmers to understand and define correctly. Further, our rule-based language has a natural correspondence with the traditional way in which syntax-directed type rules are specified.
- JAVACOP makes it possible for user-defined types specified using annotations to interact with Java's existing type system. This is a natural expectation for user-defined type systems; for example, consider the `@NonNull` annotation. A user may want to ensure that this annotation is only employed on declarations of variables or fields of reference type, rather than of primitive type. To achieve this, the AST abstraction over which JAVACOP rules work includes full Java type information about program entities, including information about type parameterization arising from Java 1.5's generics. The AST abstraction also includes other information that depends on types, for example the set of methods that a given method overrides. User-defined rules can then easily access this information. Operationally, JAVACOP enforces user-defined constraints only after regular Java typechecking has occurred.
- Since Java includes subtyping as a fundamental part of its type system, it is expected that user-defined types will also naturally have interesting relationships to one another and to existing types. For example, a user may want to disallow a `@Confined` type `S` from being considered a subtype of an unconfined supertype `T`, in order to prevent values of type `S` from escaping their package through that supertype. The JAVACOP rule language includes special facilities that allow users to easily define these kinds of relationships.
- A key feature of Java's type system is *modularity*: each compilation unit can be typechecked in isolation, given only "interface" information about directly-mentioned types (e.g., declared supertypes and the types of fields). This information can be gleaned from bytecode, without access to the associated Java source. All other types need not be present for the class to be properly typechecked and can therefore be easily changed after the fact. JAVACOP maintains the exact same modularity as Java's type system: user-defined rules are enforced by JAVACOP on each compilation unit individually, given only interface information about depended-upon types. In this way, JAVACOP integrates naturally with existing Java compiler frameworks.

  Checking each compilation unit individually does have some liabilities. For example, an apparently simple rule such as *"MyPackage must contain exactly one class"* cannot be checked with JAVACOP, because JAVACOP will never see the entire package, only the individual classes of which it is comprised. Given JAVACOP's place in the development life-cycle and the kind of syntax-directed local rules JAVACOP is designed to check, we consider modularity a valid design tradeoff. Further, in some cases rules can be reformulated so that JAVACOP can check them modularly (e.g., JAVACOP can enforce the rule *"classes in MyPackage must be named MyClass"*, and Java's naming rules will ensure that MyClass is unique).

We have implemented the JAVACOP framework as two logically independent components. First, we have extended Sun's `javac` compiler, which is written in Java, with a Java API that provides a clean interface to the compiler's internal AST and type structures. Just before code generation, a new pass in the compiler executes Java code that employs this API in order to enforce constraints on the program being compiled. Second, we have implemented a component that translates constraints in the JAVACOP rule language to Java code employing our new API. This separation of concerns allows the JAVACOP rule language to easily evolve without requiring modifications to the `javac` compiler, and it admits the possibility of easily incorporating multiple languages for enforcing constraints on programs.

We have validated the JAVACOP framework in several ways. First, we illustrate the framework's expressiveness and practicality by replicating a number of interesting type systems from the literature, including confined types, scoped types, types for race detection, nonnull types, types for immutability, and generic ownership types [49, 51, 27, 5, 47, 44]. We have also implemented several style and convention checkers, including an EJB3.0 verifier [21], as well as various code pattern detectors [18, 31] and metrics tools [13]. Finally, we have tested the performance of the JAVACOP framework. In the worst case, the addition of JAVACOP rule checking increases total compile time by a factor of two.

The rest of this paper is structured as follows. Section 2 introduces the design of JAVACOP's rule language through a number of examples. Section 3, as well as the appendix, present additional examples illustrating the flexibility of the framework. Section 4 provides details on the framework's implementation. Section 5 discusses related work to JAVACOP, and Section 6 concludes and discusses future directions.

2. The JavaCOP Rule Language

This section describes JAVACOP's rule language in detail. The first part describes some of the features of the AST representation of programs that the language employs. The rest of the section describes the features of the JAVACOP rule language, and their utility in implementing pluggable type systems, via several examples.

2.1 The Abstract Syntax Tree

The AST of a Java program is made up of linked nodes representing the program's structure: classes, methods, blocks, statements, expressions, identifiers, etc. JAVACOP's AST is an abstraction of `javac`'s AST and contains more than 50 separate node types, all subclasses of the abstract superclass `Tree`. Figure 1 lists a selection of the main AST nodes and the Java code they represent. As usual, each node provides methods to access its sub-nodes.

JAVACOP's traversal of the AST occurs as a pass after typechecking has occurred. This allows JAVACOP rules to make use of type information, which is critical for many kinds of type exten-

Tree subclass	Name	Java example
`Apply`	Method call	`meth(args)`
`Assign`	Assignment	`x=y`
`ClassDef`	Class definition	`class X{ .. }`
`Ident`	Identifier	`foo`
`If`	Conditional	`if(cond).. [else ..]`
`MethodDef`	Method definition	`void foo(){..}`
`NewClass`	Instance creation	`new World("hello")`
`Return`	Return statement	`return false;`
`Select`	Selection	`s.field`
`Skip`	Empty statement	`{;}`
`TopLevel`	Compilation unit	
`TypeCast`	Cast	`(String)s`
`VarDef`	Variable declaration	`String s = "hello";`

Figure 1. A selection of AST nodes classes and their meanings

sions. Every node in the AST contains a `type` field of type `Type`, which is set during the typechecking pass and represents the Java type of the expression represented by the node. These types include class types (which may be parameterized), array types, method types, and (bounded) type parameters; Java interfaces are represented by class types internally. Types have methods that allow their component types to be accessed, and class types have methods that allow their direct superclass (`supertype()`) and superinterfaces (`interfaces()`) to be retrieved.

In order for a class to be compiled, `javac` requires information about each nonlocal identifier — package, class, interface, method, and field name — that is referenced in the class. If `javac` were a whole-program compiler, each identifier could simply be linked to the AST node for its associated definition. However, given `javac`'s modular style of compilation, the source of some depended-upon program entities, and hence AST nodes for those entities, may be unavailable. Instead, `javac` reconstructs the necessary information about nonlocal entities from their bytecode representations, and stores it as `Symbol` objects.

The JAVACOP AST includes an abstraction of this notion of symbols, thereby allowing users to modularly access information about nonlocal identifiers in their rules. For example, a `ClassSymbol` contains the relevant information about a nonlocal class, including its supertypes, modifiers, metadata, and the types of its accessible methods and fields. For uniformity, every identifier, whether to a local or nonlocal entity, has access to a `Symbol` object representing the entity being identified. Additionally, the AST nodes representing classes, methods, etc., as well as the corresponding kinds of `Type` objects, have a method `sym()` that provides access to the associated `Symbol`.

While JAVACOP's AST abstraction is currently built on top of the AST representation in `javac`, there is a separation between the two. For example, JAVACOP's AST nodes include several useful methods that are not directly available in the underlying `javac` nodes, but instead get translated to the appropriate chain of method invocations. Because of this separation, we hope to be able to move the JAVACOP system to another compiler without changing the rule language. We plan to explore the integration of JAVACOP with the Eclipse framework.

2.2 Rule Language Overview

Figure 2 presents the syntax of the JAVACOP rule language. A pluggable type system is implemented in JAVACOP as a set of *rules*, which constrain classes via the AST representation described in the previous subsection. Each rule is declared to apply to a particular kind of AST node and provides constraints on the usage of that node, depending on type information, user-defined annotations, and other context information available at that node. To enforce user-defined rules, JAVACOP performs a depth-first traversal of the AST of a given compilation unit (i.e., a Java file). As each node is traversed, any rules that apply to that type of node are evaluated on that node.

⟨*RuleFile*⟩ ::= (⟨*Rule*⟩ | ⟨*Declaration*⟩)+
⟨*Rule*⟩ ::= 'rule' ⟨*Identifier*⟩ '(' ⟨*Joinpoint*⟩ ')' ⟨*StatementList*⟩ ⟨*FailureClause*⟩?
⟨*Declaration*⟩ ::= 'declare' ⟨*Identifier*⟩ '(' ⟨*VarDefList*⟩ ')' ⟨*StatementList*⟩
⟨*StatementList*⟩ ::= '{' ⟨*Statement*⟩+ '}'
⟨*JoinPoint*⟩ ::= ⟨*VarDef*⟩
| ⟨*Identifier*⟩ '<:' ⟨*Identifier*⟩ '@' ⟨*Identifier*⟩
| ⟨*Identifier*⟩ '<<:' ⟨*Identifier*⟩
⟨*VarDefList*⟩ ::= ⟨*VarDef*⟩ (',' ⟨*VarDef*⟩)*
⟨*VarDef*⟩ ::= ⟨*TypeIdentifier*⟩ ⟨*Identifier*⟩
⟨*FailureClause*⟩ ::= ':' ('error' | 'warning') '(' ⟨*ValueExpression*⟩ ',' ⟨*Expression*⟩ ')'
⟨*Statement*⟩ ::= ⟨*Condition*⟩
| ⟨*Quantification*⟩
⟨*Condition*⟩ ::= ('where' | 'require') '(' (⟨*VarDefList*⟩ ';')? ⟨*Expression*⟩ ')' ⟨*ConditionRest*⟩
⟨*ConditionRest*⟩ ::= ⟨*FailureClause*⟩? ';'
| ⟨*StatementList*⟩ ⟨*FailureClause*⟩?
⟨*Quantification*⟩ ::= ('forall' | 'exists') '(' ⟨*VarDef*⟩ ':' ⟨*Expression*⟩ ')' ⟨*StatementList*⟩ ⟨*FailureClause*⟩?

Figure 2. A subset of the JAVACOP Syntax. Expression syntax is not presented here, but handles most Java expressions and additionally supports declarative type testing and pattern matching.

As shown in Figure 2, there are three kinds of "join points" for JAVACOP rules, which determine when a rule is applicable to an AST node, and these are detailed in the next two subsections. We then discuss the various kinds of constraints that may be employed within a rule body, the `declare` syntax for defining helper predicates for use within a rule, and JAVACOP's facilities for error reporting.

2.3 AST Rules

The first kind of JAVACOP rule in Figure 2 is a function that starts with the keyword `rule` and includes a name, a single parameter whose type is a (subtype of) `Tree`, and a body containing a sequence of constraints. When JAVACOP's AST traversal visits a node, the node is passed to each rule that takes an argument of the node's type. For example, the `checkNonNull` rule defined in Section 1 will be passed each node representing a Java assignment statement during JAVACOP's traversal of an AST. As a result, each assignment statement will be forced to satisfy the constraints in the body of `checkNonNull`. There are several forms of constraints, which are discussed in detail in later subsections.

As mentioned earlier, JAVACOP's AST abstraction provides full type information about program entities, which is often useful for implementing pluggable type systems. As a simple example, the following rule ensures that the `@NonNull` annotation is only used for variables of reference type.

```
rule checkNonNull2(VarDef v){
 where(requiresNonNull(v)){
  require(!v.type.isPrimitive()):
   error(v,"@NonNull can only annotate"
    +" variables of reference type");
}}
```

2.4 Subtype Rules

A hallmark of most object-oriented type systems is the notion of subtyping. An important part of static typechecking ensures that this subtyping relation is respected, such that values of a given type can only ever be "viewed" as that type or a supertype. For example, the type of the right-hand side in an assignment statement must be a subtype of the type of the location being assigned, and the actual arguments to a method call must be subtypes of the corresponding formal argument types.

Pluggable type systems for a language like Java naturally require the ability to extend the existing subtyping relation, in order to prescribe the ways in which the new user-defined type specifications interact with other types, both Java types and user-defined ones.

JAVACOP's rule language supports the declarative specification of user-defined subtyping relationships. For example, as mentioned earlier, the goal of a type system for confinement [49] is to ensure that instances of types that are marked as *confined* are only accessible within the type's defining package. To achieve this goal, such a type system imposes a number of restrictions on confined types. The rule below enforces one such restriction, which disallows a value of confined type from being viewed "at" an unconfined supertype. The `confined` predicate used in the rule checks whether a given type is annotated with `@Confined`.

```
rule checkConfined(a <: b @ e){
 where(confined(a)){
  require(confined(b)):
   error(e, "Confined type "+a
    +" may not be treated as a subtype of"
    +"unconfined type "+b);
}}
```

The only syntactic difference from the AST rule syntax described earlier is the parameter list of the form `Type <: Type @ Tree`. The rule applies to any AST node that (implicitly or explicitly) changes the type of an expression, including assignment statements, parameter passing, result returning, and type casts. A rule in this style takes as arguments the original type of an expression (`a` in the above rule), the type to which the expression is being changed (`b`), and the expression itself (`e`). During an AST traversal, the `checkConfined` rule will be applied to every node that changes the view of an expression's type. At each such node, JAVACOP retrieves the old and new types as well as the expression in question and passes them to the rule. A node may have multiple such expressions (for example, a method call with multiple arguments), leading to multiple calls of `checkConfined` when that node is traversed.

As mentioned in the introduction, JAVACOP allows users to define two main kinds of type specifications, those that annotate type declarations (e.g., `@Confined`) and those that annotate variable and field declarations (e.g., `@NonNull`). The `<:` construct described above is suitable for the first kind of type specification but not the second, since the latter kind of specification is not a property of a type but of a particular variable or field. Therefore, we introduce an alternate syntax of the form `Tree <<: Symbol` for expressing subtype constraints on those kinds of type specifications. For example, the rule below subsumes the `checkNonNull` rule in Section 1 by ensuring a potentially null expression can never be viewed as `@NonNull`.

```
rule checkNonNull(node <<: sym){
 where(requiresNonNull(sym)){
  require(definitelyNonNull(node)):
   error(node, "Possibly null expression "+
               node+" considered @NonNull");
}}
```

The above rule is applicable to the same AST nodes as is the earlier `checkConfined` rule, except for type casts.[1] Rather than comparing two Java types, however, rules of this new kind are passed the AST node for the expression being stored as well as the `Symbol` for the variable or field that is being updated. At an assignment node, for example, the right-hand-side expression is tested against the `Symbol` for the variable or field being assigned.

2.5 Constraints

The body of a JAVACOP rule consists of a sequence of constraints. The basic kind of constraint has the form "**`require`**`(<condition>);`". Such a constraint is satisfied if the associated condition evaluates to `true`; otherwise the constraint fails. As a simple example, the rule below shows that JAVACOP can encode the semantics of Java's `final` modifier for classes, which requires a `final` class to have no subclasses.

```
rule finalClass(ClassDef c){
 require(!c.supertype().sym().
          hasAnnotation("Final"));
}
```

The rule checks each class definition to ensure that the class does not inherit from a class that has the `@Final` attribute — `hasAnnotation` is a built-in method on symbols for looking up a Java 1.5 attribute. A notable feature of this rule is its ability to be enforced modularly: the global property that a `@Final` class must have no subclasses is ensured by individual checks on each class in a program. These checks employ the rule language's `Symbol` objects to access "interface" information about a class's declared superclass. The JAVACOP rule language ensures that rules can be enforced modularly via the API that it provides to users. For example, while it is possible to access a given class's superclass, it is not possible to access all subclasses of that class.

To make the rule language intuitive for Java programmers, conditions include ordinary Java `boolean` expressions. These expressions can invoke methods on any AST nodes, types, and symbols in the scope of the constraint. JAVACOP also supports conditions that declaratively perform type tests and structural pattern matching, which are discussed in the next subsection.

As a syntactic convenience, JAVACOP allows a method with zero arguments to be accessed like a field. For example, the requirement above could be written more succinctly `!c.supertype.sym.hasAnnotation("Final")`. JAVACOP also includes support for Java's primitive types and `String`, and their associated operations and methods, as well as for a `List` type provided by `javac`. Finally, conditions may also employ values of two new types: a *traversal environment* `Env` holds information about the tree context surrounding a given node, and a *global environment* `Globals` is a repository for global constants, such as the type objects for `java.lang.Object` and primitive types, and the symbols for the root and empty packages. An instance of each of these two types is implicitly in scope in each rule, with the name `env` and `globals` respectively. Some examples in the next section illustrate their usage.

Often a constraint should only be applied under certain circumstances. This can be accomplished through the `where` constraint, which is similar to Java's `if` statement. Like `require`, a `where` constraint has a guard expression and a body containing a sequence of other constraints; the `where` constraint is satisfied if either its guard evaluates to `false` or all constraints in the body evaluate to `true`. An example `where` constraint is shown in the `checkNonNull` rule from the previous subsection. In that rule, the right-hand-side

[1] Because of limitations in Java 1.5 metadata, a type cast cannot include a user-defined type specification, so type casts are irrelevant.

```
@Confined class C {
  protected C() { ... }
  public @Anon String toString() { ... }
}
```

Figure 3. A confined class `C` with an anonymous `toString` method.

expression of an assignment need only be shown to be nonnull if the left-hand-side variable or field is annotated with `@NonNull`.

2.6 Pattern Matching and Type Tests

Type rules often depend on the ability to deconstruct the expressions, types, and environments that they constrain, so it is imperative that a language designed for encoding pluggable types also have this ability. To that end, JAVACOP supplies two new operators: `<-` for general type testing, and `=>` for pattern matching on AST nodes.

An expression of the form `v <- e` evaluates to true if the value of `e` at run time is an instance of the type of the variable `v` or a subtype. In that case, `e` is cast to the type of `v`, which is then assigned this value. Otherwise, the type-test expression evaluates to false. An expression may be preceded by a list of variable declarations to be bound within that expression. For example, the following constraint fails if the value of `tree.owner` is not a subtype of `ClassSymbol`:

```
require(ClassSymbol cs; cs <- tree.owner){...}
```

As another example, type systems for confinement depend upon a notion of *anonymous* methods, which do not allow `this` to be stored. Only anonymous methods may be invoked on a confined object outside of a confined scope. Figure 3 illustrates a confined class that uses an annotation on a method declaration to specify that the method is anonymous. The following rule uses `<-` to determine whether a method call within an anonymous method declaration properly preserves anonymity. The rule requires that method calls with an (explicit or implicit) receiver of `this` are either to `static` methods or to other anonymous methods. Type tests are used to distinguish between the explicit and implicit receiver cases.

```
rule anonCallsAnon(Apply a){
 where(env.enclMethod != null &&
       anonymous(env.enclMethod)){
  where(Select s; s <- a.meth){
   where(Ident i; i <- s.selected){
    where(i.name.equals("this")){
     require(s.isStatic || anonymous(s)):
      error(i, "'this' may not be used to "+
            "call non-anonymous methods.");
  }}}
  where(Ident i ; i <- a.meth){
   require(i.isStatic || anonymous(i)):
    error(i, "implicit 'this' may not be"+
            " used to call non-anonymous"+
            " methods.");
}}}
```

JAVACOP also includes an expression sublanguage for pattern matching on AST nodes. Pattern matching allows for declarative testing of properties of an AST node, while also deconstructing the node and giving names to its component nodes for use in the rest of a constraint. A pattern match is a `boolean` expression: `e => [ pat ]`. In this expression, `e` is an arbitrary expression of type `Tree`, and the pattern match succeeds if the value of `e` can successfully be matched against the pattern `pat`.

Patterns are written as fragments of Java code which must be structurally equivalent to the provided expression in order for the match to succeed. Patterns may include wildcard elements, which are written `%` to match any subtree, `*` to match any identifier name, and `...` for any number of elements in a sequence, such as statements in a block or parameters to a method. For example, the pattern `[v => [@NonNull % *(...)]` requires that the `Tree` node `v` must be a method declaration which is annotated `@NonNull`, has any return type, any name, and any number of arguments.

Patterns may also bind `Tree`, `String` (identifier name), or `List` subcomponents encountered in the pattern structure to fresh variables declared in the constraint. Variable binding involves an implicit type test: for a variable binding to succeed, the type of the component value must meet the declared type of the new variable. For example, variables can be bound to the subexpressions in the pattern described above as follows:

```
where(Tree typ, String nam, List<Tree> args ;
        v => [@NonNull typ nam(args)]){...}
```

Patterns may also test whether a given subtree is the same (by reference equality) as an expression in the current scope of the constraint. For example, if `n` is an existing variable of type `Tree`, then the following constraint requires `n` to be the first statement in its enclosing block: `require(n.parent(env) => [{`n`;...}])`

Pattern matching can significantly improve the readability and shorten the length of rules that would otherwise require multiple type tests. For example, the `anonCallsAnon` rule above can be rewritten in a single constraint using pattern matching as follows:

```
rule shortAnonCallsAnon(Apply a){
 where(env.enclMethod != null &&
       anonymous(env.enclMethod)){
  where(a.meth => [this.*] || a.meth => [*]){
   require(a.meth.isStatic ||
           anonymous(a.meth)):
    error(i, "'this' may not be used to "+
          "call non-anonymous methods.");
}}}
```

2.7 Quantification

JAVACOP supports a form of quantification over two kinds of data structures. First, constraints may universally or existentially quantify over `javac` `List`s. The syntax is similar to the syntax of the enhanced `for` in Java 1.5. For example, the following rule enforces Java's requirement that a final method not be overridden:

```
rule finalMethod(MethodDef m){
 forall(Type supertype :
    m.enclClass.transitiveSupertypes){
  forall(Symbol other :
     supertype.sym.memberLookup(m.name)){
   where(other.isFinal){
    require(!m.sym.overrides(other,
                             supertype.sym))
     :error(m, "Overriding final method");
}}}}
```

The outer `forall` iterates over a list of all transitive supertypes of the given method's enclosing class, binding each to the name `supertype` in turn. The inner `forall` then iterates over a list of all members of `supertype` that have the same name as the given method. We require the variable used to bind each element of a list to have the same type as the declared element type of the list. The syntax for existential quantification is identical, except that it uses the keyword `exists` instead of `forall`.

The example above also illustrates another advantage of the fact that JAVACOP's AST representation contains full type information. The `overrides` method provided by the `MethodSymbol` class determines whether one method overrides another. This functional-

ity is provided by `javac` and is well-defined after the typechecking pass, so the JAVACOP API simply accesses it when desired by users. If JAVACOP's AST abstraction did not have access to type information, users would be forced to manually check whether a method overrides another, which is nontrivial given features like method overloading, privacy modifiers, and covariant return types.

Second, JAVACOP allows quantification to be used to iterate over all nodes in a depth-first traversal from a given AST node. During this traversal, only nodes that match the declared type of the quantifier variable are considered. For example, the following rule requires that every local variable in a method be assigned at least once. The rule uses universal quantification to iterate over every variable declared in the given method and existential quantification to find at least one assignment to this variable.

```
rule allLocalVarsMustBeAssigned(MethodDef m){
 forall(VarDef v : m){
  where(v.init == null){
   exists(Assign a : m){
    require(a.lhs.getSymbol == v.getSymbol);
   }:error(v,"Variable " + v +
             "is never assigned")
}}}
```

2.8 Auxiliary Predicates

In addition to rules, JAVACOP allows users to declare auxiliary predicates, analogous to the auxiliary predicates sometimes used in formal type systems (e.g., Featherweight Java [38]), using the `declare` keyword. These predicates are not tested directly during JAVACOP's AST traversal but instead are used simply as helpers for rule definitions. Predicates are invoked by the bodies of rules and other predicates using a traditional function-call syntax. For example, the rule `checkNonNull` from Section 2.4 makes use of a `requiresNonNull` predicate defined as follows:

```
declare requiresNonNull(Tree t){
 require(t.holdsSymbol
  && t.getSymbol.hasAnnotation("NonNull"));
}
```

This predicate gets the given node's associated symbol if it has one and uses the symbol to check whether the node has the appropriate annotation. Auxiliary predicates provide the usual benefits of procedural abstraction. In this case, the `requiresNonNull` predicate serves to cleanly separate the logic that determines how "nonnullness" is annotated from the logic that determines the behavior of programs employing such an annotation. This separation makes it easy to later augment or modify the annotation mechanism, for example to use a marker interface instead of Java's metadata facility.

Rule and predicate bodies naturally support a form of conjunction for constraints, by sequencing multiple constraints. Predicates themselves additionally provide a declarative form of disjunction. Similar to predicates in Prolog, JAVACOP allows an auxiliary predicate to have multiple definitions; an invocation of the predicate succeeds if at least one of the definitions' bodies is satisfied.

For example, the `checkNonNull` rule from Section 2.4 makes use of the `definitelyNotNull` predicate, which checks whether a given `Tree` object is definitely nonnull. It is natural to define this predicate with a case analysis on different subtypes of `Tree`, by providing multiple definitions of the predicate. A few representative definitions are provided below.

```
/* A node is nonnull if annotated as such */
declare definitelyNotNull(Tree t){
 require(requiresNonNull(t));
}
```

```
/* The value of an assignment is nonnull
   if the value being assigned is nonnull */
declare definitelyNotNull(Assign a){
 require(definitelyNotNull(a.rhs));
}

/* Class instantiation is never null */
declare definitelyNotNull(NewClass n){
 require(true);
}
```

A `declare` implicitly performs a type test on a given node against the declared type of its argument. This type test desugars into a `require` constraint: the `declare` definition fails if the type test fails. For example, the last definition above fails if the given node does not represent a Java `new` expression.

2.9 Error Reporting

As in any type system, it is important to provide programmers useful feedback about rule failures that occur during checking of their programs. Precise failure reporting is even more critical in the context of a pluggable type system, since programmers will be less familiar with the checks being performed than they would be for a fixed type system. Informative messages can also make it easier for type-system designers to debug their rules through testing as they are being developed.

To this end, JAVACOP supports user-defined *failure clauses*, which define the message to be reported when a rule fails. A failure clause has three components: an indication of whether an error or warning is to be emitted, via the keywords `error` and `warning`, respectively; the source position at which the error is to be reported, which may be a `Tree` object or the value `globals.NOPOS`; and an expression containing the message to report. An error causes compilation to halt, while a warning allows compilation to continue normally. In this way, JAVACOP users can easily decide how rule violations are to be treated on a case-by-case basis. For example, a "pure" pluggable type system as defined by Bracha [8] might treat all violations as warnings, while a use of pluggable types to enforce a form of security might treat particular violations as fatal errors.

A failure clause can be applied to a constraint or rule simply by appending "`:<errortype>(<position>,<message>)`" to it. For example, the `finalMethod` rule from Section 2.7 provides a failure clause for the nested `require` constraint, using the source position of the `MethodDef` object `m` as the position at which to report the error. If this `require` fails during JAVACOP's AST traversal, an error will be reported at the source position, using the given string as the error message.

A failure clause can be placed on any constraint, no matter how nested it is within compound constraints. As a single exception, we disallow failure clauses within an `exists` constraint. The reason is simply that the failure of a constraint within an `exists` does not necessarily result in failure of the entire constraint, so it would not make sense to signal a failure at that point.

When a constraint fails to be satisfied, JAVACOP searches for the nearest enclosing failure clause and executes it. For example, the following example signals a warning if `a` or `b` fails but signals an error if `c` fails.

```
require(a){
  require(b);
  require(c):error(pos1, "error - c failed");
}:warning(globals.NOPOS,
            "warning - a or b have failed")
```

3. Case Studies

To illustrate the benefits of the JAVACOP framework for pluggable type systems, we have implemented JAVACOP rules for a variety of interesting and nontrivial type systems from the research literature. These systems include:

- a type system for confinement [49]
- a type system for *scoped types* [51], which provide safe region-based memory management in Real Time Java
- a checker for static detection of race conditions [27]
- a checker for structured object aliasing via *islands* [36]
- a checker for nonnull references
- a checker for Javari's notion of reference immutability [5, 47]
- a type system for Generic Ownership (the core of the OGJ language) [44]

We have also implemented JAVACOP rules that enforce other kinds of program constraints and check for various program properties, including:

- an EJB3.0 verifier [21]
- two checkers supplied by the PMD Java checker [18]
- rules that identify classes matching the *Degenerate Classes* micro patterns [31]
- rules that gather information for Chidamber and Kemerer's object-oriented metrics [13].

In this section, we demonstrate JAVACOP's expressivity and usability by describing the salient features of a number of these checkers, focusing on the type systems. More examples can be found in the appendix of this paper.

3.1 Confined Types

Confined types were introduced to create static restrictions on how references may be shared among objects [49]. The goal of a type system for confinement is to ensure that instances of confined types are only accessible within their defining package. To meet this goal, confined types must obey several restrictions: they cannot be public; the cannot inherit from unconfined classes (other than `java.lang.Object`); and Java language constructs (such as casts) must be restricted. As mentioned in Section 2, confinement also depends on a notion of *anonymous* methods.

JAVACOP's implementation of a type system for confinement requires 12 rules and 18 predicate declarations. We have already presented some of the basic rules in Sections 2.4 and 2.6. We describe a few more of the key rules here.

Confined Class Declaration

Confined types cannot be public, nor can they belong to the default Java package. This is encoded in the following rule:

```
rule ConfinedExample1(ClassDef c){
 where(confined(c)){
  require(!c.isPublic()):
   error(c, "Confined class may not be public");
  require(c.pkg() != globals.emptyPackage):
   error(c, "Confined class may not be " +
            "in the default package");
}}
```

This rule is defined in terms of an auxiliary `confined` predicate, which checks whether a type is declared to be confined. This predicate could be defined to check for an annotation like `@Confined`, to check for a particular marker superinterface, or even to check for a particular naming convention on types.

Confined Class as Extension

A confined class may not extend an unconfined type. This prevents inherited fields and methods leaking a confined instance. The JAVACOP rule that enforces this is:

```
rule ConfinedExample2(ClassDef c){
 where(confined(c)){
  forall(Type s : c.supertypes()){
   require(confined(s) || s == globals.objectType):
    error(c, "A confined class may not extend "
             + "an unconfined superclass");
}}}
```

This rule shows again how to develop rules that utilize the inheritance hierarchy. Via universal quantification, it states that all supertypes, including both superclasses and superinterfaces, of a confined class must be confined. While the `supertypes` method does not return all transitive supertypes of `c`, it is sufficient for this rule because if all direct supertypes are confined, by induction, all transitive supertypes must be confined as well. As a single exception, a confined class may have `java.lang.Object` as a superclass, since this cannot be avoided in Java.

Confined Type Coercion

To maintain the integrity of the confined type system, coercion from a confined type to an unconfined type is disallowed. The following rule, seen previously, enforces the necessary behavior using JAVACOP's subtype join point.

```
rule ConfinedExample3(a <: b @ pos){
 where(confined(a)){
  require(confined(b)):
   error(pos, "confined type "+a+
              " may not be cast to "+
              "unconfined type "+b);
}}
```

As discussed in Section 2.4, this rule applies to all AST nodes where an (implicit or explicit) type coercion occurs. Because of the `ConfinedExample2` rule, the only way that `b` can be unconfined is if it is the class `java.lang.Object`. The above rule enables the notion of confinement to coexist safely with Java's choice to have a single root class in the language.

Confined Access via Anonymous Methods

Earlier in the paper, we presented the notion of anonymous methods for confined classes and showed a rule that helps to enforce anonymity of such methods. The benefit of anonymous methods is that it is always safe for them to be invoked on instances of confined types. This is not true of non-anonymous methods, as they could potentially leak `this`. The following JAVACOP rule requires every invocation of a non-anonymous method on a confined object to occur within a confined scope:

```
rule ConfinedExample4(Apply a){
 where(!anonymous(a.meth) && !a.meth.isStatic){
  where(Tree receiver; a.meth => [receiver.*]){
   where(confined(receiver)){
    require(confined(a.meth.enclClass())):
     error(a,"Non-anonymous method '"+
              a.meth.sym+
              "' called on confined class '"+
              receiver.sym+"'");
}}}}
```

The rule declaratively dispatches on all method invocations (the `Apply` AST node) and uses pattern matching to access the receiver in the method invocation.

3.2 Scoped Types

Scoped types [51] use a variant of confinement to provide a form of safe region-based memory management for Real Time Java. All types defined in a package declared to be *scoped* are implicitly considered to be scoped, except for a distinguished *gate class* that provides controlled access to the package's types from outside the package [2]. We have implemented a type system for scoped types with five rules and 19 declarations. We present a selection of these rules and declarations that illustrate JAVACOP's utility.

Scoped Packages and Scoped Types

The following `declare` returns true if a package is declared to be scoped. By convention, a package is scoped if the package is nested within either the `imm` or `scope` packages.

```
declare scopedPackage(PackageSymbol p){
 require(List<Symbol> o; o <- p.transitiveOwners){
  require(o.length >= 1 &&
          (o.head.name.equals("imm") ||
           o.head.name.equals("scope")));
}}
```

The next `declare` determines whether a type is scoped. The first definition states that a type is scoped if it does not match the naming convention for a gate type and it is in a scoped package. The second definition says that a type is considered scoped if it has a generic parameter that is a scoped type.

```
declare scoped(Type t){
 require(!gateNamed(t.sym));
 require(scopedPackage(t.pkg));
}

declare scoped(Type t){
 exists(Type p : t.allparams){
  require(scoped(p));
}}
```

The second definition above takes advantage of JAVACOP's ability to reason about Java generics. In this case, the list of type parameters of a type can be easily checked for a specific property.

The following `rule` uses the helper predicates defined above to ensure that the "visibility" of a scoped type is respected. Except for a few special cases (captured by `safeNode(t)` below), a variable of scoped type can only be accessed in the type's defining package and nested packages.

```
rule scopedTypesVisibility(ClassDef c){
 forall(Tree t : c){
  where(t.type != null && !safeNode(t)){
   where(scoped(t.type)){
    require(t.type.pkg
              .isTransitiveOwner(c.pkg)):
     error(t, "Scoped type visible only in its "+
                "package.");
}}}}
```

Scoped Type Widening

The visibility rule presented above can only be statically verified if it is impossible to remove or change a variable's scope. If this were not the case, a value of some type `T` which should not be visible in a package `P` could be made visible in `P` simply by casting the value to a type whose scope is visible in `P`. The following rule ensures that a variable's scope is not changed by disallowing a type in a scoped package from being coerced implicitly or explicitly to a type in another package. JAVACOP's subtyping rule form allows this semantics to be naturally encoded:

```
rule scopedTypesCasting(a <: b @ pos){
  where(scopedPackage(a.pkg) {
   require(a.pkg == b.pkg):
    warning(pos, "Scoped type "+a+" may not be"+
                 "widened to a type in another"+
                 " package.");
}}
```

3.3 Race Condition Detection

Flanagan and Freund proposed a type system for detecting race conditions in Java [27]. In this type system, each field of a class is declared to have another object as its *lock*. The type system then ensures that a field's associated lock is held (via Java's `synchronized` construct) whenever the field is accessed. We have implemented a pluggable version of such a type system in JAVACOP, utilizing Java 1.5 metadata to annotate fields with their associated locks. For simplicity, we assume that a field's lock is either `this` or another field declared in the same class. A field's lock is specified via a `@LockedBy` annotation, which includes the name of the lock as an associated value. Our implementation also properly handles `requires` clauses, which represent method preconditions; we implement them via `@Requires` annotations on method declarations.

Immutable Locks

For soundness, Flanagan and Freund require the expressions used as locks to be immutable. The following rule ensures this property by requiring that a field used as a lock is declared `final`. The `hasLockSpec` helper predicate determines whether a field has a properly formatted `@LockedBy` attribute.

```
rule finalLock(VarDef v){
 forall(VarDef v2 : env.enclClass){
  where(hasLockSpec(v2.sym)){
   where(Literal n; v2 => [@LockedBy(n) % *]){
    where(n.value.equals(v.name)){
     require(v.isFinal)
      :warning(v2,v.name+" must be immutable.");
}}}}}
```

The rule makes use of pattern matching to easily pull the lock name out of a field declaration.

Field Access When Lock Held

Ensuring that field accesses are properly synchronized requires that information on the associated lock is properly retrieved from the type system. Since `@LockedBy` is an annotation on a field declaration, it can be found on the `Symbol` that represents the field. The following rule signals a warning if a field access does not occur within a `synchronized` block guarded by the field's associated lock.

```
declare checkLockAccess(Select field, String name){
 exists(Tree t: env.enclClass.pathTo(field)){
  require(Synchronized s; s <- t){
   require(s.lock.getFullName.equals(name));
}}}

rule selectSynchCheck(Select s){
 where(hasLockSpec(s.sym)){
  where(Constant c; c <- s.sym.attributes
                         .getCompound("LockedBy")
                         .getComponent("value")){
   require(
     checkLockAccess(s,
       s.selected.getFullName+"."+c.fullName)):
    warning(s, "Unsafe access of "+s.name);
}}}
```

Because JAVACOP is integrated with the Java type system, all necessary information can be retrieved from the type and symbol information that is available modularly at a field access. For example, the above rule works properly for field accesses that occur outside of the field's declaring class, even if only the bytecode (but not the source) of that class is available. The `checkLockAccess` predicate above makes use of JAVACOP's `Environment` class to access the context surrounding a field access. Existential quantification iterates over every block in this context to find an appropriate `synchronized` statement. Although not presented here, `checkLockAccess` also allows the needed lock to be found in the `@Requires` clause of the enclosing method.

3.4 Nonnull

In Section 2, we showed several rules governing the use of a `@NonNull` annotation. This section illustrates a more sophisticated rule that provides a simple form of flow sensitivity.

Assuming that threading does not exist or is handled correctly, it is always safe to treat a variable or field as `@NonNull` directly after a null check has been performed. For example, the following code should not be marked as an error.

```
if(x != null){
  @NonNull Object nonnull_x = x;
  ...
}
```

The rules discussed previously, however, are not sufficient to recognize the assignment to `nonnull_x` as safe. The following rule resolves this limitation.

```
rule checkNonNull(Assign a){
 where(declaredNonNull(a.lhs)){
  require(definitelyNotNull(a.rhs)
          || safeNullableAssign(a)):
   error(a,"Assigning null value to @NonNull "+
           "variable.");
}}

declare safeNullableAssign(Assign a){
 require(localVariable(a.rhs));
 require(Tree l ;
           a.parent(env).parent(env)
            => [if(l != null){`a`;...}]
       || a.parent(env)
            => [if(l != null)`a`]){
  require(l.sym == a.rhs.sym);
 }
}
```

The above rule generalizes the rule for assignments in the introduction, in order to allow another possible way in which the right-hand side of an assignment can be considered nonnull. The `safeNullableAssign` predicate uses the environment to access the enclosing lexical scope of the assignment node along with pattern matching to test whether this context performs the appropriate null check. The first pattern matches the case when the assignment is the first statement in a block, while the second pattern matches the case when the assignment is the only statement guarded by the `if`. Whichever pattern matches the context provides the value for `l` to be used by the innermost `require`.

3.5 Reference Immutability

Javari [5, 47] is a Java extension that introduces a `readonly` modifier. Javari's type system ensures that any reference marked `readonly` is transitively immutable. We have implemented a significant subset of Javari in JAVACOP using JAVACOP annotations and rules to emulate the behavior of the `readonly` modifier. We present a selection of the rules and end with a discussion of the compromises required to implement Javari with an annotation-based system.

Transitive Readonly

The following predicates are a selection of those defining the read-only property. Read-only references include those fields annotated as `@Readonly`, the `this` reference within a method that is marked as `@ReadonlyThis`, and the inductive case of any reference obtained by dereferencing a field (that is not marked as `@Mutable`) of a read-only reference. These predicates demonstrate JAVACOP's ability to use procedural abstraction and recursion to describe complex structural properties simply.

```
declare readonlyS(Symbol s){
 require(s.hasAnnotation("javari.Readonly"));
}

declare readonly(Tree t){
 require(!t instanceof Indexed);
 require(t.holdsSymbol && readonlyS(t.sym));
}

declare readonly(Select s){
 require(!s.isStatic
         && !mutable(s)
         && readonly(s.selected));
}

declare readonly(Ident i){
 require(i.name.equals("this")
         && readonlyThis(env.enclMethod));
}

declare readonly(Ident i){
 require(!i.isLocal
         && !i.isStatic
         && !mutable(i)
         && readonlyThis(env.enclMethod));
}
```

Enforcing Immutability

To prevent modification of state reached via a read-only reference, it is necessary to constrain the operations that can alter state, such as assignments and method calls. The following rule is an example of such a constraint. It prevents assignment to array elements and non-static fields accessed via a read-only reference.

```
rule readonlyAssignment(Assign a){
 where(!a.lhs.isStatic){
  where(Tree x; a.lhs => [x.*]){
   require(!readonly(x)
           || assignable(a.lhs)):
    warning(a, "Assigning to field of"+
               " read-only reference "+x);
  }
  where(a.lhs => [*] && !a.lhs.isLocal){
   require(!readonlyThis(env.enclMethod)
           || assignable(a.lhs)):
    warning(a, "Assigning to field of"+
               " read-only this");
 }}
 where(Tree x; a.lhs => [x[%]]){
  require(!readonly(x)):
   warning(a, "Mutating read only array"+x);
}}
```

Pattern matching allows us to validate cases without complex casts or sequences of method invocations. The first clause uses a simple pattern `[x.*]` to match field assignments, and it ensures that

either the receiver is read-only or the field is marked assignable. The second clause matches bare identifiers without an explicit receiver. These may be either fields of the `this` reference or local variables, so the latter case must be excluded by requiring that the assignee is not local. In the former case, the rule requires that either the enclosing method is marked as `@ReadonlyThis` or the field is marked assignable. The third clause matches assignment to an array index as an lvalue, which must be forbidden if the reference to the array is read-only.

Readonly Value Flow

The value flow rule for `@Readonly` is the inverse of that for the nonnull example seen previously. Whereas a nonnull system constrains what values may acquire the `@NonNull` annotation, the readonly system permits any value to acquire the `@Readonly` modifier but constrains the ability to remove it. The following rule describes this constraint; the value of any expression which has been deemed read-only may only flow into a position marked with the `@Readonly` modifier.

```
rule readonlyValueFlow(expr <<: sym){
 where(readonly(expr)){
  require(readonlyS(sym) ||
          (thismutable(expr) &&
           assignToThsMutLHS(expr.parent(env)))):
   warning(expr, "cannot cast away readonly");
}}
```

The predicate declaration `assignToThsMutLHS` describes the one case where a read-only value may be assigned to a field without the `@Readonly` modifier: when an assignable this-mutable field (a field which is marked neither read-only nor mutable, and therefore inherits its mutability from the reference used to access it) is assigned the value of a this-mutable field of the same object.

```
declare assignToThsMutLHS(Tree t){
 require(Tree lhs, Tree rhs;
          t => [lhs = rhs]){
  require(assignable(lhs));
  require(thismutable(lhs));
  require(sameSelected(lhs,rhs));
}}
```

Implementation Compromises

Several compromises were necessary to implement Javari using annotations, and without modifying Java's semantics. Tschantz and Ernst [47] describe in an appendix several problems with the approach of using annotations, each of which we had to address. We discuss a subset of these here.

First, Javari uses the `readonly` modifier on method declarations to identify two different properties of the method:

- That the return type of the method is read-only.
- That the `this` reference is to be treated as read-only within the method body.

These uses are differentiated by the location of the modifier: before the return type to modify the return type and immediately before the method body to modify the `this` reference. Java 1.5 annotations only support the former syntax. Therefore it was necessary for us to choose another annotation to indicate that the `this` reference is to be considered read-only. We use `@ReadonlyThis` for that purpose.

Second, using the `@Readonly` annotation alone, it is not possible to specify that the contents of an array are to be considered read-only, while the array itself is mutable. A similar problem arises with the use of generic collections — since annotations cannot be applied to uses of types (or type parameters), our system cannot specify the mutability of generic parameters. We mitigate this problem through a new annotation `@ReadonlyContent`, which indicates that the contents of an array or a reference to a type extending `Collection` is read-only. The `@ReadonlyContent` property applies transitively across array indexing and retrieval using `.get(%)` from collections. Additionally, the `toArray(...)` methods of `Collection` classes result in `@ReadonlyContent` values, and the common non-mutating methods `get`, `indexOf`, `contains*`, `isEmpty` and `size` are marked as `@ReadonlyThis`.

This behavior is enforced by the following predicates, along with a definition and value-flow rule for the `readonlyContent` predicate that are analogous to those for `readonly`.

```
declare readonly(Indexed i){
 require(readonlyContent(i.indexed));
}

declare readonly(Apply a){
 require(Tree x; a => [x.get(%)]){
 require(extendsCollection(x.type));
 require(readonlyContent(x));
}}

declare readonlyContent(Apply a){
 require(Tree k; a => [k.get(%)]){
 require(extendsCollection(k.type));
 require(readonlyContent(k));
}}

declare readonlyContent(Apply a){
 require(Tree t; a => [t.toArray(...)]){
  require(readonlyContent(t));
  require(extendsCollection(t.type));
}}
```

Finally, because method overloading is based on Java's type system alone, without consideration of annotations, JAVACOP rules cannot alter the behavior of method overloading and therefore are unable to implement this component of Javari.

4. Implementation

Fig. 4 gives an overview of the structure of the system: JAVACOP takes a Java program and checks it against a set of constraint rulesets, producing errors and warning messages where the constraints are not met. Because JAVACOP incorporates a full Java compiler (Sun's `javac`) it also produces compiled Java bytecode. This is (at least psychologically) important, because it means that JAVACOP errors have some teeth: if a class causes a ruleset to raise an error, then JAVACOP will not generate code for that class. Of course, this design choice can be easily changed without affecting the rest of the system.

The JAVACOP system has two major components. First, JAVACOP augments the `javac` compiler with JAVACOP.Framework, an object-oriented framework that provides a clean interface to the compiler's internal AST and type structures [1]. Second, the system includes the JAVACOP.Compiler component, which compiles constraints written in the JAVACOP rule language into Java code that uses the JAVACOP.Framework API to access a program's AST. The JAVACOP.Framework loads the rulesets compiled by JAVACOP.Compiler and, just prior to code generation, invokes each ruleset in turn on the fully-attributed AST of each Java class it is compiling.

In this section, we will highlight three important consequences resulting from JAVACOP's implementation design.

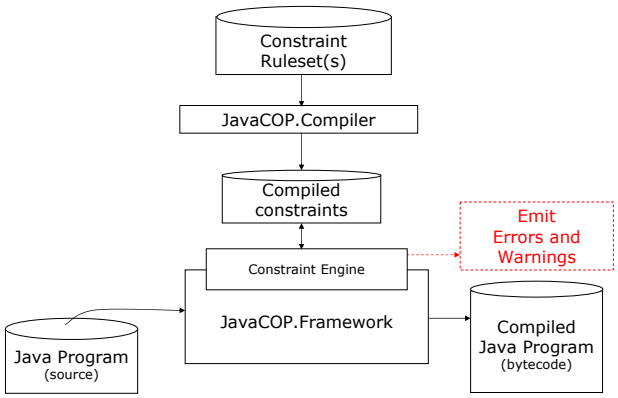

Figure 4. An overview of JAVACOP's implementation

4.1 Checking Source Code

The first consequence of JAVACOP's design is that it checks Java source code, rather than bytecode. This imposes some limitations: JAVACOP can only check programs for which source code is available, and parsing source code is less efficient than reading in bytecode. The key advantage, though, is that much more information is present in Java source code than the bytecode — in particular, generic type information and many annotations are only present in the source code. As described earlier, JAVACOP can still glean interface information from bytecode, which is necessary to check the source code for a class against the compiled code for depended-upon types.

4.2 JAVACOP.Framework API

The second consequence stems from the JAVACOP.Framework offering a stand-alone API into the compiler's data structures: programmers can write Java classes that use this API directly — although our experience of using this API directly strongly motivated us to design the JAVACOP rule language. This separation, however, allowed us to develop and test the rule compiler independently from the program checker. The JAVACOP.Framework API also insulates the compiled rulesets from any ongoing changes to the Java compiler's implementation. For space reasons, a more detailed description of this framework is presented elsewhere [1].

4.3 Performance

Thirdly, to demonstrate that JAVACOP is a practical implementation, we measured its performance compiling a range of sample programs using a range of rulesets. The sample programs include several well-known open source examples, the Java code generated by the rules mimicking PMD behavior in JAVACOP (described in the appendix), and a real time program intended to be used with Scoped Types from the OVM project [43]. We compiled each program first with no rulesets, then with the Scoped Types, PMD, Confined Types, Nonnull and Micro Patterns rulesets individually, and then as a "torture test" with JAVACOP checking all these rulesets simultaneously.

The measurements were taken on a Dell Optiplex GX250, with an Intel Pentium IV 2.5GHz and 1GB RAM, running NetBSD 3.99.11. JAVACOP was run using the Sun Java HotSpot(TM) Client VM (build 1.5.0_05-b05, mixed mode, sharing). Each test was run five times, and timed using the `time` built-in command from the `bash` shell. The wall clock ('real') time was measured, the highest and lowest values were discarded, and the remaining three averaged to produce the final figure. Because the Confined Types and Scoped Types rules rely on annotations not present in most of the examples, we modified these rulesets so that *every* class would be checked as if it were confined or scoped, respectively.

Fig. 5 presents the results. For each configuration the upper row is time in seconds, while the lower row is time relative to the "No Rules" condition — in each case lower is better. The key point from this table is that in the worst case, performing JAVACOP rule checking takes less than twice the time of compiling without any checking. Closer inspection shows that most of this time is spent in the Scoped Types ruleset: this is not surprising as those rules (a development of Confined Types discussed above) make very detailed checks, since they are designed to support real-time programming. The other rulesets, including Confined Types themselves, impose a compilation time overhead of between one and fifteen percent. We have not performed any optimisations on JAVACOP, nor have we investigated any kind of incremental compilation support. Nonetheless, these performance numbers demonstrate the practicality of JAVACOP's design.

5. Related Work

The term "pluggable types" was coined recently by Gilad Bracha [8] to describe optional type systems that can be layered on top of existing languages without affecting those languages' dynamic semantics. Bracha suggests that pluggable types should allow type systems to evolve independently of the underlying language, and that programmers could choose which pluggable type system(s) to use with their programs, depending on the programs' requirements. JAVACOP provides the first practical framework designed for introducing pluggable types in Java, allowing programmers to use and implement a range of type systems and other tools, without the costs of building a full compiler, parser, or type checker. In this section we discuss a range of work related to JAVACOP, and to pluggable types more generally.

Program Structure Constraints The earliest systems related to pluggable types were not designed as type systems per se, but rather to express syntactic constraints over programs, often to check programming style (like a customisable Lint). CCEL [22], the C++ Constraint Expression Language, was one of the first of these languages. CCEL constraints are written in a clean syntax, reminiscent of the relational calculus, and can be grouped into rulesets that are checked and managed as groups. The Java Tools Language (JTL) [17] is a more recent system with many of the same features as CCEL (except for using Java rather than C++, and with an execution model based on Prolog), motivated by Gil and Maman's corpus analysis of Java code [31]. JTL's syntax is particularly noteworthy, designed to be as close to Java's syntax as possible to make it easy for programmers to define constraint rules. JQuery [39] also uses Prolog to express structural constraints over Java programs, but rather than checking constraints it uses them to define complex browser views. JSR269 [20] defines an API for processing annotations. Annotation processors can check constraints and also generate new Java classes (e.g., RMI or EJB stubs) which are then themselves checked.

Compared to our work, CCEL, JTL, and JQuery have a restricted data model, containing primarily classes, methods (functions and virtual functions), and fields; JQuery also provides access to Eclipse's dependency structures. These systems do not "look inside methods" or traverse abstract syntax trees, so while they can express constraints on the program structures, they cannot describe type constraints within method bodies. JSR269 annotation processors are more powerful as they are written in Java (rather than a

System	Classes	No ruleset	Scoped	PMD	Confined	Nonnull	Micro	All
Hello World	1	0.972	1.053	1.014	1.052	1.016	1.015	1.078
		1.000	**1.083**	**1.043**	**1.082**	**1.045**	**1.044**	**1.109**
PMD ruleset	7	1.674	1.977	1.737	1.727	1.676	1.672	2.075
		1.000	**1.181**	**1.038**	**1.032**	**1.001**	**0.999**	**1.240**
OVM example	197	3.534	5.242	3.874	3.918	3.621	3.661	5.642
		1.000	**1.483**	**1.096**	**1.109**	**1.025**	**1.036**	**1.596**
JAVACOP	368	4.977	8.341	5.400	5.637	5.157	5.098	9.276
		1.000	**1.676**	**1.085**	**1.133**	**1.036**	**1.024**	**1.864**
PMD	443	12.435	16.289	13.091	13.338	12.688	12.688	17.229
		1.000	**1.310**	**1.053**	**1.073**	**1.020**	**1.020**	**1.386**
Jython	630	6.362	10.240	7.202	7.302	6.595	6.642	11.360
		1.000	**1.610**	**1.132**	**1.148**	**1.037**	**1.044**	**1.786**
JEdit	805	7.687	12.683	8.708	8.831	7.907	8.107	14.255
		1.000	**1.650**	**1.133**	**1.149**	**1.029**	**1.055**	**1.854**

Figure 5. JAVACOP Compilation Times

domain-specific language like JAVACOP) and can both check and generate code. On the other hand, JAVACOP's language is designed specially to support checking types and annotations, and JAVACOP (and its JAVACOP.Framework API) provides access to the program's AST, unlike JSR269.

Abstract Syntax Tree Constraints PMD [18] is a widely-used system for enforcing constraints on the abstract syntax tree of Java programs. Constraints can be specified either as classes implementing a *visitor* pattern, or an XPATH expression. CheckStyle [11] is another much-used system, similar to PMD, also based on XPATH. Compared with JAVACOP, these systems do not contain type or symbolic information and therefore support only syntactic constraints. Most of the examples we have presented in this paper, which we feel are representative of the kinds of rules that would be developed for pluggable type systems, are not expressible in these above systems because of the need for symbol and type information. Furthermore, Java code and XPATH queries are less readable than JAVACOP's declarative rules, and much further from most specifications of type systems. A clear example of the improved readability that pattern matching affords can be found in the appendix.

An older system, ASTLOG, [19] provides a domain-specific language for writing queries over the ASTs of C or C++ programs. ASTLOG is again based on a variant of Prolog; the target program's structure and AST are made accessible via the Prolog database. Compared with JAVACOP, ASTLOG's language is clearly more powerful in regards to flow-sensitivity and temporal properties, but like other systems which load whole programs into databases, it is non-modular and less efficient than JAVACOP. ASTLOG also does not have support for type system access or for any of the more recent Java features such as annotations and generics. Thus, it is not an ideal framework in which to implement the kinds of pluggable type systems seen in this paper.

More recently, CodeQuest [34, 33] has extended JQuery to provide a comprehensive Prolog-based program querying system. Like JAVACOP and ASTLOG, CodeQuest can in principle express constraints across types and methods, but JQuery requires the whole program to be stored in a relational database and then efficiently optimises recursive queries to that database. The JunGL system [48] carries on from CodeQuest by incorporating a functional language to define program refactorings. Compared with JAVACOP, CodeQuest certainly provides much better support for ad-hoc queries across entire projects, and has a highly sophisticated implementation; JunGL adds full access to data and control flow information. On the other hand, JAVACOP is tightly focused on defining pluggable type systems and checkers, and implements them modularly and straightforwardly within traditional compiler technology.

Eichberg, Schäfer and Mezini [24, 23] describe how a whole-program XML database, XIRC, including a typed AST, can be used to check constraints on programs. A program is loaded into the database, and then constraints are expressed in the form of queries in the XQuery query language. XIRC is more powerful than JAVACOP as its database can include any development artifact — so, for example, it can check EJB 2.0 components for conformity with their XML deployment descriptor, or with documentation in Word imported into XML. On the other hand, JAVACOP constraints are arguably more readable than XML queries, and are certainly closer to the kind of rules used in existing descriptions of pluggable type systems. JAVACOP's checking is also completely modular.

Static Analysers FindBugs [37] identifies *bug patterns* — code which indicates bugs or potential bugs — in Java programs by examining their bytecode. FindBugs is one example of an increasing set of bug-finding static analysis tools. The rules enforced by FindBugs are written in Java and largely fixed — it is not designed as a user-extensible tool. On the other hand, this allows FindBugs to implement relatively sophisticated custom analyses. Several extensible static analysers for C (e.g., [35, 4]) allow users to express rules as state machines that are enforced by a path-sensitive dataflow analysis. These systems can express requirements that are not expressible in JAVACOP, due to its minimal support for flow sensitivity. At the same time, these systems do not naturally express the kinds of syntax-directed pluggable type rules that JAVACOP targets.

Aspects and Meta-Objects Shomrat Yehudai [45] has described how AspectJ can enforce architectural constraints, such as *"an expensive operation must not be called from within a performance critical section of code"*, by defining pointcuts on the static call graph. AspectJ supports a range of structural constraints, but the granularity of its pointcuts seems too coarse to easily express the restrictions necessary for many pluggable type systems [30].

Inasmuch as JAVACOP layers a new type system on top of an extant base language, it is related to reflexive or meta-object systems, such as OpenJava [46]: Bracha and Ungar provide a good overview of design issues in such systems [9]. General reflexive systems are often much more powerful than JAVACOP, providing dynamic access to the structures of running programs, and allowing those programs to be modified at runtime [12]. On the other hand, compared with JAVACOP, reflexive systems provide weak support for analysing or checking the fine details of types and expressions within method bodies.

Type Systems CQual [28, 29] is a system for user-defined *type qualifiers* for C programs. Qualifiers are analogous to the Java 1.5 type annotations that JAVACOP employs. JAVACOP's rule language is much more expressive than that of CQual, which only allows users to specify subtyping relationships among qualifiers. However, CQual supports flow sensitivity and a form of qualifier inference, both of which JAVACOP lack.

The Clarity framework [14, 15] extends the idea behind CQual by providing an explicit language for user-defined qualifier rules. This language allows specification of simple patterns on individual expressions, but it does not allow the kinds of rich constraints on larger-scale program entities that JAVACOP supports. Clarity also allows programmers to provide an invariant defining the semantics of a qualifier, and an automatic theorem prover validates user-defined rules with respect to this invariant. JAVACOP lacks any form of rule validation.

JAVACOP is also loosely related to the research on soft typing [10]. Soft typing systems use type inference to add types to programs in untyped languages such as Scheme [50] or Erlang [40], inserting explicit runtime type checks where the program does not conform to the static type scheme. In contrast, pluggable type systems check explicit type annotations, do not alter the target program, and can raise hard errors if the types fail to check. Further, pluggable type systems are designed to be user-extensible, while soft type systems typically employ a fixed type inference system.

Finally, it is interesting to compare JAVACOP to extensible compiler frameworks such as JastAdd [25] and Polyglot [41]. Extensible compilers are certainly more powerful than JAVACOP, supporting arbitrary (and arbitrarily complex) analyses and extensions to syntax and semantics as well as type systems. Such systems, however, come at the cost of a much more general (and thus much larger and more complex) overall system. Where JAVACOP produces rulesets that simply "plug-in" to a standard Java compiler, these systems produce whole new compilers. JAVACOP's focus on separate, independent rulesets makes it straightforward to check multiple JAVACOP rulesets simultaneously, while combining multiple language extensions in Polyglot, say, is a much more subtle endeavour [42].

6. Conclusions

In this paper we have presented the first practical framework designed for implementing pluggable types for Java. Our contributions include:

- The design of the JAVACOP rule language for declaratively specifying natural and modular type constraints on Java programs.
- An implementation of JAVACOP, which consists of a compiler for the rule language and a checker for the compiled constraints in an extended Java compiler.
- A validation of JAVACOP through the implementation of several pluggable type systems (and other constraint sets), and through performance experiments.

We plan a range of future extensions to JAVACOP. We hope to incorporate direct support for a form of flow sensitivity, which will increase the range of annotations that can be checked by JAVACOP and increase the precision of checking for existing annotations like `@NonNull`. We are also planning to extend the error clauses so that rules can count the number of times a constraint fails, or to accumulate lists of failing program elements, rather than simply producing error messages — features of many other constraint system verifiers.

To increase the efficiency of checking complex rulesets on large programs, we are planning to cache quantification and predicate results — since JAVACOP is a declarative language, such caching will suffer from no problems with side effects, but can provide many of the benefits of imperatively assigning extra attributes to AST nodes. More pragmatically, the JAVACOP rule compiler is currently written in Haskell; we plan to port this to Java. Note that the extended `javac` compiler is already written in Java.

Finally, we plan to continue to develop JAVACOP rulesets to support a wide range of type annotations, style checkers, and analysis tools, to bring the benefits of pluggable types to Java programmers everywhere.

Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant Nos. CCF-0427202 and CCF-0545850; by a generous gift from Microsoft Research; by an IBM Eclipse Innovation Grant; by the EPSRC grant Practical Ownership Types for Objects and Aspect Programs, EP/D061644/1; and by the Royal Society of New Zealand Marsden Fund. Thanks to Alex Potanin, Gilad Bracha, Brian Foote, Itay Maman, Jan Vitek, John Potter, and the anonymous reviewers from OOPSLA and ECOOP 2006 for comments on prior drafts.

Further Case Studies

In this appendix, we present a number of additional examples of systems implemented with JAVACOP, as mentioned in Section 3.

Enterprise JavaBeans

As part of the J2EE platform, the Enterprise JavaBeans (EJB) architecture was created to aid in the development of component-based distributed software. The EJB architecture requires applications to satisfy a number of well-formedness constraints, which are detailed in the EJB specifications [21]. These specifications are very large and complicated, so J2EE implementations typically provide a verifier tool that checks some rules to avoid accidental deviations from the specification. Since these rules are all predicated on the use of EJB annotations, the EJB specification itself can be viewed as a large, self-contained type system.

As a test of JAVACOP's expressiveness, we have implemented many of the rules from the EJB 3.0 specification [21]. In particular, we were able to express in JAVACOP all the rules that have been expressed by other checker systems [24, 11], except for two types of rules. First, a few rules inherently require global program knowledge, for example knowledge of all call sites of a given method. Since JAVACOP is a modular checker, all possible information about future references for fields and methods may not be available for a class when it is initially being checked. Any rules that would require such information are not possible to write in JAVACOP. Second, some rules depend on the temporal order of method calls at run time, while JAVACOP's rule language is flow-insensitive.

All in all, we implemented 73 JAVACOP rules and 140 predicate declarations. A few of the more interesting rules are presented below.

Proper Use of `@TransactionAttribute`

The first example is a rule to enforce the part of the EJB 3.0 specification that states "The `TransactionAttribute` annotation can only be specified if container managed transaction demarcation is used" [21]. What the specification is stating is that any time the annotation `TransactionAttribute` is used, there may not be an annotation argument to any annotation that assigns the value `BEAN` to `transactionManagement`. Here is an example of a Java class that violates this rule:

```
@Stateless(transactionManagement=BEAN)
```

```
  @TransactionAttribute(MANDATORY)
    class ExampleBean { ... }
```

The following JAVACOP rule and predicate enforce this constraint:

```
rule EJBExample1(ClassDef c){
 where(c.sym.
       hasAnnotation("TransactionAttribute")){
  require(!isTransactionManagementBean(c.mods)):
   warning(c,"Bean cannot be annotated with "
             +"@TransactionAttribute and "
             +"have bean-managed "
             +"transactions.");
}}

declare isTransactionManagementBean(Modifiers m){
 exists(Annotation a: m){
  exists(Tree arg: a.args){
   where(Tree lhs, Tree rhs; arg => [lhs = rhs]){
    require(lhs.getName.
                equals("transactionManagement") &&
            rhs.getName.equals("BEAN"));
}}}}
```

The `isTransactionManagementBean` predicate helps determine whether a class represents a transaction management bean, using pattern matching to deconstruct the name-value pairs in the class modifiers list, while the rule itself checks for the `TransactionAttribute` attribute on such beans.

Public Getters for Entity Bean Fields

For entity bean classes, the EJB 3.0 specification states "[F]or every persistent property *property* of type *T* of the entity bean, there is a getter method named `getProperty`" [21]. This is necessary to access the fields since persistent fields in entity bean classes must either be declared private or protected. The following rule enforces such behavior:

```
rule EJBExample2(ClassDef c){
 where(isEntity(c)){
  forall(VarDef v: c.varDefs){
   require(hasGet(c,v)):
    error(v, "Field "+v.name+" does not have "
             +"a getter method.");
}}}

declare hasGet(ClassDef c, VarDef v){
 exists(MethodDef m: c.methodDefs){
  where(m.name.equals(
        "get"+v.name.charAt(0).toUpper
        +v.name.substring(1,v.name.length))){
   require(m.sym.isPublic);
}}}
```

Of particular note in this rule is JAVACOP's ability to utilize Java's libraries to perform transformations on the field's name. In this way, the proper getter method's name can be enforced.

Business Method Exceptions

The EJB 3.0 specification says that "the methods of [a] business interface should not throw the `java.rmi.RemoteException`" [21]. The rule to enforces this discipline is shown here, although predicate definitions are omitted:

```
rule EJBExample3(ClassDef c){
 where((c.sym.hasAnnotation("Stateful")
        || c.sym.hasAnnotation("Stateless"))
       && !hasSessionInterface(c)){
  forall(Type supr: c.supertypes){
   where(supr.isInterface){
    forall(Symbol m: supr.sym.methods){
     forall(Type excp: m.thrown){
      require(!hasGivenType(excp,
                "java.rmi.RemoteException")):
       warning(c,"Business Interface "
                +supr.fullName
                +" defines method "+m.name
                +" throwing RemoteException");
}}}}}}
```

Since business interfaces are not determined by an annotation, the first `where` in this rule determines whether the interfaces implemented by a (stateful or stateless) session bean class are business interfaces. In the case that they are, the `throws` clause of each method of each business interface is then checked for the presence of the forbidden exception. Perhaps the most interesting feature of this rule is that it will check whether previously existing interfaces, which may predate EJB specification, are safe to use as business interfaces.

Generic Ownership

Generic Ownership [44] is a type system that combines generic types and object ownership [16]. By building on an existing parametrically polymorphic language framework, Generic Ownership makes only minimal changes to an existing language to provide ownership, while simultaneously providing an ownership-based language that is type-generic. We used JAVACOP to implement a version of the Generic Ownership language OGJ. Because Generic Ownership depends heavily on parametric types, this implementation demonstrates that JAVACOP can support sophisticated extensions to parametric type systems. The full implementation is around 770 lines in 43 rules and predicates; in this section we present an overview of some of the key rules.

Classes require owner parameters

In OGJ, most programmer-defined classes are required to take an extra generic parameter to record their owner. This "owner parameter" must be declared to extend the class `World` (`ogj.World` in our implementation), and must be the last parameter provided, as in:

```
class List<Element, Owner extends World> { ...
```

In JAVACOP, we can enforce this behaviour as follows:

```
rule RequireOwnerParameter(ClassDef c){
 where(c.type.supertype == globals.objectType){
  require(hasOwnerParameter(c.type) ||
          isOwnerType(c.type)):
   error(c, "Toplevel class ("+c.type+")
             needs owner parameter");
 }
 where(hasOwnerParameter(c.type.supertype)){
  require(c.type.typarams.last ==
          c.type.supertype.typarams.last):
   error(c,"Ownership not preserved when "+
          c.type+" extends "+c.type.supertype);
 }
 where(hasOwnerParameter(c.type)){
  require(c.type.supertype == globals.objectType
          || hasOwnerParameter(c.type.supertype)):
   error(c,"Owned type extends non-owned type");
 }
}
```

This rule carries out a simple case analysis in its `where` clauses. Classes extending `java.lang.Object` must declare a new owner parameter, while classes that extend another class must monotonically preserve their superclass's owner parameter. Finally, we require that classes may not be declared with an owner parameter

unless they extend either `Object` or a superclass with an owner parameter.

Preservation of ownership

The class formation rule above ensures that OGJ classes will always have an owner parameter. We then need to ensure that this information cannot be lost from an object's static type — that is, we have to prevent OGJ classes being cast to Java's raw types or to `java.lang.Object`. For example, consider the following generic list of books: its elements are public `Book` objects owned by `World`, while the `List` itself is private, owned by `This`. Note that `World` and `This` are constants instantiating the owner parameters of `List` and `Book` [44].

```
List<Book<World>,This> l =
    new List<Book<World>, This>;

Object obj = l; // loses ownership information
List rawList = l; // so do raw types
```

In JAVACOP, this rule can be implemented as follows:

```
rule preventRawCasting(sub <: supr @ e){
 where(hasOwnerParameter(sub)){
  require(!supr.isRaw):
   error(e, "Cast "+sub+" to raw type "+supr);
  require(supr != globals.objectType):
   error(e,"Cast "+sub+" to Object");
}}
```

This rule, like similar rules for Confined Types and other related systems, uses JAVACOP's subtyping rule form (see Section 2.4) to check all explicit and implicit casts. If the cast is from a type with an owner parameter, we forbid that type from being cast to a raw type or `Object`.

Deep ownership

These two rules are sufficient to ensure that every OGJ object has an owner that is known statically. OGJ however has a more structured ownership model, known as "deep ownership", that establishes a transient owners-as-dominators property on the heap [44]. This means that private data (e.g., data owned by `This`) must not be stored in publically-accessible structures. For example, code such as:

```
List<Book<World>,This> l = new ...;
```

that declares a private list of public `Book` objects is quite permissible, but the alternative:

```
List<Book<This>,World> l = new ...;
```

must be prevented. In JAVACOP, this rule requires some quite complex constraints on generic class instantiation — especially because owners can be represented by type parameters such as (but not limited to) the `Owner` parameter of every OGJ class.

Following the OGJ formal system [44], we consider each class definition, and (using JAVACOP's `forall` quantifier over a tree) recursively check that all types present within the class are well formed for deep ownership.

```
rule wellFormedTypes(ClassDef c){
 forall(Tree t : c){
  where(t.type instanceof ClassType
     && !t instanceof ClassDef){
   require(wellformed(t.type)):
    error(t,"Type "+t.type+" of "+t+"
     not well formed for deep ownership");
}}}
```

The bulk of the work then is done by the `wellformed` predicate. First, types that don't take parameters (mostly primitive types, `void`, `null`, and packages) are considered well formed, as are type variables.

```
declare wellformed(Type t){
 require(unparameterizable(t));
}

declare wellformed(TypeVar v){
 require(true);
}
```

Second, for types that do not have an owner parameter (presumably Java class types that are being used in OGJ programs), we check that all their generic type parameters (`typarams`) are well formed.

```
declare wellformed(ClassType c){
 require(!hasOwnerParameter(c));
 forall(Type prmtype: c.typarams){
  require(wellformed(prmtype));
}}
```

Finally we check that each generic type instantiation is well formed:

```
declare wellformed(ClassType c){
 require(Type ownerofC;
     ownerofC <- c.typarams.last){
  forall(Type prmtype: c.typarams){
    where(isOwnerType(prmtype)){
     require(
       isDeepOwnerSubtype(ownerofC, prmtype));
    }
    where(Type ownerofPrmType;
      ownerofPrmType <-
               prmtype.classBound.typarams.last){
      where(isOwnerType(ownerofPrmType)){
       require(isDeepOwnerSubtype(ownerofC,
                                  ownerofPrmType));
   }}}}
 forall(Type prmtype: c.typarams){
   require(wellformed(prmtype));
}}
```

This rule fetches the class's owner (`ownerofC`) from the instantiation of the last (ownership) generic type parameter, and then iterates over each of the type parameters. If the parameter is a "naked" owner parameter (i.e., it extends `World`) then we check that the parameter is outside the class's owner (via the auxiliary `isDeepOwnerSubtype` predicate); alternatively if the parameter is an OGJ class, we fetch its owner (`ownerofPrmType`) in turn from *its* last parameter, and check that that owner is valid. The `isDeepOwnerSubtype` predicate directly encodes the OGJ ownership subtyping relationship [44]. To finish, we again recurse through all the type parameters.

Instance encapsulation

These three rules work together to provide and statically enforce a dynamic model where objects are deeply nested inside each other. The final rule uses this structure to enforce strong object encapsulation: private data may only be accessed from within their owners. For example, the following code declares an encapsulated list within the `Catalogue` class:

```
class Catalogue<Owner extends World>{
 List<Book<World>,This> myList;

 int volumes() {return myList.size();}
}
```

This list can be accessed freely inside the current instance of the class but cannot be used outside it, because types owned by `This` may only be accessed (explicitly or implicitly) via `this`.

```
Catalogue cat = ...;

...cat.myList.size();
 // error as myList is private.
```

In JAVACOP, properly expressing this rule requires handling a number of cases. We begin with a rule that catches Java's "." dot selection operator via JAVACOP's `Select` AST node, and then simply delegates to the JAVACOP `validThisSelect` predicate:

```
rule OGJThisSelect(Select s){
 require(validThisSelect(s)):
  warning(s,"Invalid select on "+s.type);
}
```

The `validThisSelect` predicate clauses catch various forms of dot operator. First, we permit any calls via `this` and any uses of the dot operator to construct compound types:

```
declare validThisSelect(Select s){
 require(s => [this.*]
         || s.sym instanceof TypeSymbol);
}
```

Next, we check each case of the Select node — these clauses catch method sends and variable accesses — and require that their generic type arguments do not include `This` via the `thislessType` predicate.

```
declare validThisSelect(Select s){
 require(s.type instanceof MethodType){
  require(thislessType(s.sym.type));
}

declare validThisSelect(Select s){
 require(s.sym instanceof VarSymbol){
  require(thislessType(s.sym.type));
}
```

Finally, we have a series of predicates that recursively define a "thisless" type as being a type which does not involve the owner `This` in any of its generic type parameters.

```
declare thislessType(Type t){
 require(unparameterizable(t));
}

declare thislessType(ClassType c){
 require(!(c.fullName.equals("ogj.This")));
 forall(Type t: c.allparams){
  require(thislessType(t));
}}
```

The full OGJ ruleset straightforwardly extends these cases to cover Java's other types and selection constructs.

PMD style checking rules

PMD is a popular utility for Java that scans application code for code patterns that match stylistic rules [18]. Such rules encode behavior for finding potential bugs and suboptimal code. However, these rules can be difficult to create and many are difficult to understand.

JAVACOP is fully capable of recreating these rules in a more readable and declarative fashion. To show this, we implemented the *basic* and *finalizers* rulesets from PMD, which describe purely structural properties that are considered bad style or bugs, as 23 JAVACOP rules and 12 declarations. The fine-grained error marking and reporting of JAVACOP makes reporting violations of these rules flexible and robust. We show two examples here:

Unconditional Ifs

A common coding discipline that PMD implements is that `if` statements should not be guarded by boolean literals. PMD rules may be written as Java code or in XPath, making the rules relatively tedious for programmers to understand. For example, PMD's XPath implementation of this rule is shown below.

```
//IfStatement/Expression
 [count(PrimaryExpression)=1]
 /PrimaryExpression/PrimaryPrefix
 /Literal/BooleanLiteral
```

In JAVACOP, this same rule can be implemented as follows:

```
rule PMDExample1(If i){
 require(!(i => [if(true)%])):
  warning(i.cond,"Body of this if "
                 +"always executed");
 require(!(i => [if(false)%])):
  warning(i.cond,"Body of this if "
                 +"never executed");
}
```

Although longer than the XPath rule, the JAVACOP rule is more readable and is a better approximation of a traditional type rule through its use of pattern matching. JAVACOP's error clauses also allow it to produce meaningful error messages if an unconditional `if` statement is detected.

Empty Statements

Another PMD rule focuses on empty statements, or semicolons without an associated expression. PMD considers such code — when not used in conjunction with a loop — as either redundant or a bug. The JAVACOP version of this rule is:

```
rule PMDExample2(Skip s){
 require(env.tree instanceof Block
         && skipOnlyLoop(env.next.tree, s)):
  warning(s, "Empty statement in block");
}

declare skipOnlyLoop(ForLoop l, Skip s){
 require(skipOnlyBlock(l.body, s));
}

declare skipOnlyLoop(WhileLoop l, Skip s){
 require(skipOnlyBlock(l.body, s));
}

declare skipOnlyLoop(DoLoop l, Skip s){
 require(skipOnlyBlock(l.body, s));
}

declare skipOnlyBlock(Block b, Skip s){
 require( b => [{`s`}]);
}
```

The first `require` in this rule ensures that the empty statement is directly enclosed within a `Block`, and then passes the next enclosing `Tree` to the `skipOnlyLoop` predicate. The `skipOnlyLoop` predicate dispatches on the type of the `Tree` passed to it, failing unless the `Tree` is a `ForLoop`, `WhileLoop` or a `DoLoop` node, in which cases it uses the `skipOnlyBlock` predicate to test that the `Skip` statement is the only element in the loop body. The `skipOnlyBlock` predicate uses a pattern match expression to ensure that the argument `Block b` matches a `Block` containing only

one statement, which is equal to the JAVACOP expression 's' in the current scope, that is, the `Skip` statement.

PMD's XPath equivalent for this rule, shown below, is particularly difficult to read and debug.

```
//Statement/EmptyStatement
 [not(
       ../../../ForStatement
       or ../../../WhileStatement
       or ../../../BlockStatement/ClassOrInterfaceDeclaration
       or ../../../../../../../ForStatement/Statement[1]
        /Block[1]/BlockStatement[1]/Statement/EmptyStatement
       or ../../../../../../../WhileStatement/Statement[1]
        /Block[1]/BlockStatement[1]/Statement/EmptyStatement)
 ]
```

Micro Patterns

Micro patterns [31] are commonly used coding idioms that occur at a low level of abstraction. For example, the first micro pattern, *Designator*, is defined as *an interface with absolutely no members*. While a programmer may not develop an application with these micro patterns in mind, identifying which micro patterns are being used, and how frequently, can give insight into the broader behavior of a large application.

To illustrate the utility of JAVACOP beyond detecting errors, we implemented the *Degenerate Classes* micro patterns as a set of seven rules and seven declarations, which print a warning when a micro pattern is identified. Checking an application's code with these rules will then create a data set that can be analysed further. We present a representative rule that detects three micro patterns for interfaces with no members: *Designator* (no members), *Taxonomy* (no members, but extends one interface), and *Joiner* (no members, but extends two or more interfaces). Any Taxonomy or Joiner interfaces are, by definition, also Designators.

```
rule MicroPatternsExample(ClassDef c){
 where(c.isInterface &&
       c.sym.members.length == 0){
  require(false):
   warning(globals.NOPOS,
           "Designator: "+c.flatName);
  require(c.interfaces.length != 1):
   warning(globals.NOPOS,
           "Taxonomy: "+c.flatName);
  require(c.interfaces.length < 2):
   warning(globals.NOPOS,
           "Joiner: "+c.flatName);
}}
```

Software Metrics

Chidamber and Kemerer [13] defined one of the earliest sets of metrics for object-oriented systems. The six metrics they defined include: number of methods per class; depth of inheritance hierarchy; number of children; coupling between objects; response set of objects; and lack of cohesion of methods. We have written 9 rules (with no auxiliary declarations) to support gathering all of these metrics in JAVACOP. As with the micro patterns, our JAVACOP rules issue warnings when the basic data for the metrics is encountered: an external tool (a simple shell script) can analyse this data to derive the metrics. The following pair of JAVACOP rules determines coupling between objects.

```
rule MetricsExample(Select s){
 where(env.enclClass != null){
  where(ClassSymbol otherclass;
        otherclass <- s.getSymbol){
   require(otherclass == env.enclClass.sym):
    warning(globals.NOPOS,
            env.enclClass.sym.flatName
            +" couples to "+otherclass.flatName);
}}}
rule MetricsExample(Ident i){
 where(env.enclClass != null){
  where(ClassSymbol otherclass;
        otherclass <- i.getSymbol){
   require(otherclass == env.enclClass.sym):
    warning(globals.NOPOS,
            env.enclClass.sym.flatName
            +" couples to "+otherclass.flatName);
}}}
```

These two rules join on both dot selection and identifier AST nodes. If we are in the context of a class definition and the selection or identifier node refers to a class other than that of the surrounding definition, we report that the enclosing class is coupled to the node's class.

References

[1] C. Andreae. JavaCOP — user-defined constraints on Java programs. Honours Report, Computer Science, Victoria University of Wellington, 2005.

[2] C. Andreae, Y. Coady, C. Gibba, J. Noble, J. Vitek, and T. Zhao. STARS: Scoped types and aspects for real-time systems. In *Proceedings of ECOOP 06*, 2006.

[3] K. Arnold, J. Gosling, and D. Holmes. *The Java Programming Language Third Edition*. Addison-Wesley, Reading, MA, third edition, 2000.

[4] T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–3. ACM Press, 2002.

[5] A. Birka and M. D. Ernst. A practical type system and language for reference immutability. In *OOPSLA Proceedings*, pages 35–49, New York, NY, USA, 2004. ACM Press.

[6] J. Bloch. A metadata facility for the Java programming language. Technical Report JSR 175, `www.jcp.org`, 2002.

[7] J. Boyland, J. Noble, and W. Retert. Capabilities for sharing: A generalisation of uniqueness and read-only. In *ECOOP Proceedings*, 2001.

[8] G. Bracha. Pluggable type systems. In *OOPSLA Workshop on Revival of Dynamic Languages*, 2004.

[9] G. Bracha and D. Ungar. Mirrors: Design principles for meta-level facilities of object-oriented programming languages. In *OOPSLA Proceedings*, 2004.

[10] R. Cartwright and M. Fagan. Soft typing. In *PLDI Proceedings*, 1991.

[11] Checkstyle Developers. Checkstyle. `http://checkstyle.sourceforge.net`, 2005.

[12] S. Chiba. Load-time structural reflection in Java. In *ECOOP Proceedings*, 2000.

[13] S. R. Chidamber and C. F. Kemerer. Towards a metrics suite for object oriented design. In *OOPSLA Proceedings*, 1991.

[14] B. Chin, S. Markstrum, and T. Millstein. Semantic type qualifiers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2005)*, June 2005.

[15] B. Chin, S. Markstrum, T. Millstein, and J. Palsberg. Inference of user-defined type qualifiers and qualifier rules. In *European Symposium on Programming*, 2006.

[16] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *OOPSLA Proceedings*, pages 48–64. ACM Press, 1998.

[17] T. Cohen, J. Y. Gil, and I. Maman. JTL — the Java tools language. In *OOPSLA Proceedings*, Oct. 2006.

[18] T. Copeland. *PMD Applied*. Centennial Books, Nov. 2005.

[19] R. F. Crew. ASTLOG: a language for examining abstract syntax trees. In *Proceedings of the USENIX Conference on Domain-Specific Languages*, Santa Barbara, Oct. 1997.

[20] J. Darcy. Pluggable annotation processing API, 2005.

[21] L. DeMichiel. *Enterprise JavaBeans Specification, Version 3.0*. SUN Microsystems, 2004.

[22] C. K. Duby, S. Meyers, and S. P. Reiss. CCEL: A metalanguage for C++. In *C++ Conference*, pages 99–116, 1992.

[23] M. Eichberg, M. Mezini, K. Ostermann, and T. Schäfer. Xirc: A kernel for cross-artifact information engineering in software development environments. In *Working Conference on Reverse Engineering*, 2004.

[24] M. Eichberg, T. Schäfer, and M. Mezini. Using annotations to check structural properties of classes. In *FASE*, 2005.

[25] T. Ekman and G. Hedin. Rewritable reference attributed grammars. In *ECOOP Proceedings*, 2004.

[26] M. Fahndrich and K. R. M. Leino. Declaring and checking non-null types in an object-oriented language. In *OOPSLA Proceedings*, 2003.

[27] C. Flanagan and S. N. Freund. Type-based race detection for Java. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 219–232. ACM Press, 2000.

[28] J. S. Foster, M. Fähndrich, and A. Aiken. A Theory of Type Qualifiers. In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 192–203, Atlanta, Georgia, May 1999.

[29] J. S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 1–12. ACM Press, 2002.

[30] C. Gibbs and Y. Coady. Making real-time abstractions concrete with aspects. In *The 3rd Workshop on Java Technologies for Real-time and Embedded Systems*, 2005. Held in conjunction with OOPSLA 2005.

[31] J. Y. Gil and I. Maman. Micro patterns in Java code. In *OOPSLA Proceedings*, 2005.

[32] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification Second Edition*. The Java Series. Addison-Wesley, Boston, Mass., 2000.

[33] E. Hajiyev, M. Verbaere, and O. de Moor. Codequest: Scalable source code queries with datalog. In *ECOOP Proceedings*, 2006.

[34] E. Hajiyev, M. Verbaere, O. de Moor, and K. de Volder. Codequest: Querying source code with datalog. In *OOPSLA Companion*, 2005.

[35] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. In *PLDI Proceedings*, 2002.

[36] J. Hogg. Islands: aliasing protection in object-oriented languages. In *OOPSLA Proceedings*, pages 271–285. ACM Press, 1991.

[37] D. Hovemeyer and W. Pugh. Finding bugs is easy. In *OOPSLA Companion*, 2004.

[38] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, May 2001.

[39] D. Janzen and K. D. Volder. Navigating and querying code without getting lost. In *AOSD Proceedings*, 2003.

[40] S. Marlow and P. Wadler. A practical subtyping system for Erlang. In *ICFP Proceedings*, 1997.

[41] N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An extensible compiler framework for Java. In *Proceedings of CC 2003: 12'th International Conference on Compiler Construction*. Springer-Verlag, Apr. 2003.

[42] N. Nystrom, X. Qi, and A. C. Myers. J&: software composition with nested intersection. In *OOPSLA Proceedings*, 2006.

[43] OVM/J Consortium. The open virtual machine project. http://www.ovmj.org/, 2004.

[44] A. Potanin, J. Noble, D. Clarke, and R. Biddle. Generic ownership for generic Java. In *OOPSLA Proceedings*, 2006.

[45] M. Shomrat and A. Yehudai. Obvious or not?: regulating architectural decisions using aspect-oriented programming. In *AOSD Proceedings*, 2002.

[46] M. Tatsubori, S. Chiba, M.-O. Killijiand, and K. Itano. OpenJava: A class-based macro system for Java. In *Reflection and Software Engineering*, volume 1826 of *LNCS*, 2000.

[47] M. S. Tschantz and M. D. Ernst. Javari: adding reference immutability to Java. In *OOPSLA Proceedings*, pages 211–230, New York, NY, USA, 2005. ACM Press.

[48] M. Verbaere, R. Ettinger, and O. de Moor. JunGL: a scripting language for refactoring. In *Proceedings of the 28th International Conference on Software Engineering (ICSE)*, 2006.

[49] J. Vitek and B. Bokowski. Confined types. In *OOPSLA Proceedings*, 1999.

[50] A. K. Wright and R. Cartwright. A practical soft type system for scheme. *TOPLAS*, 19(1):87–152, 1997.

[51] T. Zhao, J. Noble, and J. Vitek. Scoped types for real-time Java. In *RTSS Proceedings*, 2004.