JAVACOP: Declarative Pluggable Types for Java

SHANE MARKSTRUM
Bucknell University
DANIEL MARINO, MATTHEW ESQUIVEL, and TODD MILLSTEIN
University of California, Los Angeles
and
CHRIS ANDREAE and JAMES NOBLE
Victoria University of Wellington

Pluggable types enable users to enforce multiple type systems in one programming language. We have developed a suite of tools, called the JAVACOP framework, that allows developers to create pluggable type systems for Java. JAVACOP provides a simple declarative language in which program constraints are defined over a program's abstract syntax tree. The JAVACOP compiler automatically enforces these constraints on programs during compilation. The JAVACOP framework also includes a dataflow analysis API in order to support type systems which depend on flow-sensitive information. Finally, JAVACOP includes a novel test framework which helps users gain confidence in the correctness of their pluggable type systems. We demonstrate the framework by discussing a number of pluggable type systems which have been implemented in JAVACOP in order to detect errors and enforce strong invariants in programs. These type systems range from general-purpose checkers, such as a type system for nonnull references, to domain-specific ones, such as a checker for conformance to a library's usage rules.

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

Additional Key Words and Phrases: JAVACOP, ,pluggable type systems

Portions of this work were published in Andreae et al. [2006b].
This material is based upon work supported by the National Science Foundation under Grant Nos. CCF-0427202, CCF-0545850, and OISE-0813362; by a gift from Microsoft Research; by an IBM Eclipse Innovation Grant and an IBM Faculty Award; and by the Royal Society of New Zealand Marsden Fund.
Authors' addresses: S. Markstrum (corresponding author), Department of Computer Science, Bucknell University, PO Box A0551, 701 Moore Avenue, Lewisburg, PA 17837; email: smarkstr@cs.ucla.edu; D. Marino, M. Esquivel, T. Millstein, Department of Computer Science, University of California at Los Angeles, Los Angeles, CA 90095; C. Andreae, J. Noble, Victoria University of Wellington, New Zealand.

DOI 10.1145/1667048.1667049 http://doi.acm.org/10.1145/1667048.1667049

ACM Reference Format:

Markstrum, S., Marino, D., Esquivel, M., Millstein, T., Andreae, C., and Noble, J. 2010. JavaCOP: Declarative pluggable types for Java. ACM Trans. Program. Lang. Syst. 32, 2, Article 4 (January 2010), 37 pages. DOI = 10.1145/1667048.1667049 http://doi.acm.org/10.1145/1667048.1667049.

1. INTRODUCTION

Type systems in mainstream languages like Java [Arnold et al. 2000; Gosling et al. 2000] provide a discipline for programmers that ensures important well-formedness properties. For certain applications, extending these standard type systems with a richer set of types that enforces a stronger discipline can provide better guarantees about program safety. For example, *nonnull* types ensure variables never point to the `null` value [Fähndrich and Leino 2003], *confined* types ensure no references to instances of a class can escape that class's defining package [Vitek and Bokowski 1999], and *readonly* types ensure that an object's state is not modified through a particular reference [Boyland et al. 2001]. In addition, programmers routinely rely on disciplines that are not traditionally enforced by languages or type systems: syntactic restrictions, design patterns, and architectural styles, among others. Without language support, programmers must document desired programming disciplines informally and enforce them manually, a process which is tedious and error prone. Instead of requiring manual enforcement or a dedicated language extension, it is desirable to allow programmers to externally augment languages to enforce new disciplines. Enforcing these additional disciplines as optional type extensions has come to be known as *pluggable type systems* [Bracha 2004].

In this article, we present the design, implementation, and evaluation of a framework for pluggable type systems in JAVACOP , which we call JAVACOP (Constraints On Programs). The JAVACOP framework consists of a suite of three major tools that have been developed and enhanced over the last four years based on our experience building and using practical pluggable type systems.

— *A declarative rule language for structural constraints.* The heart of any type system is a set of syntax-directed rules that constrain program entities based on their structure, static context, and type annotations. JAVACOP provides a declarative language that is tailored for expressing such rules. While we could simply provide a visitor framework and allow users to implement traversals over an Abstract Syntax Tree (AST) data structure manually, we believe that our declarative language makes rules significantly easier to create, understand, and evolve.

— *A dataflow analysis engine for pluggable type systems.* While the core logic of a type system is flow insensitive and is naturally expressed in our declarative language, some pluggable type systems require flow-sensitive reasoning. For example, a nonnull type system should allow a possibly-null variable `x` to be considered nonnull after a successful test of the form `x != null`. Rather than generalizing the JAVACOP language to support full-blown flow sensitivity, we provide a Java API that allows developers to easily define flow analyses whose results can then be used in ordinary flow-insensitive

rules. This approach has allowed us to make use of flow-sensitive reasoning in a variety of expressive and unanticipated ways while keeping the high-level structure of each pluggable type system simple and declarative.

— *A test harness for pluggable type systems.* A set of JAVACOP rules, like other programming artifacts, can have errors. To address this problem, we have developed a novel approach to testing JAVACOP type systems. We observe that many pluggable type systems are meant to enforce a simple set of runtime invariants. For example, a nonnull type system should ensure that a `@NonNull` variable or field never has the value `null`. Our test harness allows developers to ensure that programs in a given test suite which pass the pluggable type system's checks also satisfy the intended invariants when executed. In essence, our test harness helps developers test for violations of *type soundness*.

The JAVACOP framework is implemented as an extension of the OpenJDK `javac` compiler, version 1.7.0-ea. A programmer can invoke `javac` as usual but additionally provide command-line arguments indicating the pluggable type systems to use. The associated rules are enforced on the given classes' ASTs after the standard Java typechecking pass, with all warnings and errors printed to standard output. Because JAVACOP type systems, like the base Java type system, are modular (enforced class-by-class rather than on an entire program) JAVACOP naturally scales to large applications without disrupting existing Java development processes.

We have built a diverse suite of pluggable type systems in JAVACOP and used them to ensure important invariants, enforce design patterns, and detect errors in hundreds of thousands of lines of existing Java software. This article presents the results of our experiences with four qualitatively different pluggable type systems in JAVACOP . Other uses of JAVACOP have been described elsewhere, including a pluggable type system to enforce safe memory management in the real-time specification for Java [Andreae et al. 2006a]; a pluggable type system to enforce fine-grained access control policies [Fischer et al. 2009]; and a pluggable type system to enforce safety-critical software standards that has been used by the JSR 302 working group on Safety Critical Java Technology [JSR 302 2006].

JAVACOP and all associated tools were released under the open-source GNU General Public License v2.0. The JAVACOP implementation and all example pluggable type systems in this article are available at `http://javacop.sourceforge.net`.

The rest of this article is structured as follows. Section 2 introduces the design of JAVACOP'S rule language through a number of examples. Section 3 presents our experience building and using two domain-specific pluggable type systems developed in JAVACOP'S rule language. Section 4 describes JAVACOP'S facilities for flow-sensitive reasoning and Section 5 presents our experience building and using two general-purpose pluggable type systems that leverage flow sensitivity in interesting ways. Section 6 describes and presents experiences with our pluggable type system testing framework; Section 7 presents some performance experiments to demonstrate JAVACOP'S practicality

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

Fig. 1. A subset of the JavaCOP syntax. Expression syntax is not presented here, but handles most Java expressions and additionally supports let binding and pattern matching (Section 2.6).

for interactive development. Section 8 discusses some limitations of the JavaCOP framework and ideas for resolving them; Section 9 compares with related work; and Section 10 concludes.

2. THE JavaCOP RULE LANGUAGE

This section describes JavaCOP's rule language in detail and its utility in implementing pluggable type systems. We present the language informally by example here; elsewhere the first author has presented a formal semantics of the JavaCOP language by translation to a form of Datalog with negation [Markstrum 2009]. The syntax of the JavaCOP language, shown in Figure 1, is designed to be natural for programmers already familiar with Java.

2.1 Preliminaries

The AST of a Java program (fragment) is made up of linked nodes representing the program's lexical structure: classes, methods, blocks, statements, expressions, identifiers, etc. JavaCOP's AST is an abstraction of the OpenJDK compiler AST, in which all the node types are subclasses of the abstract superclass `JCTree`. Figure 2 lists a selection of these AST nodes and the Java code they represent. Each node provides methods and fields to access its subnodes.

A pluggable type system is implemented in JavaCOP as a set of *rules*, which constrain programs via the AST representation described before. These rules are translated into regular Java code manipulating the OpenJDK's AST and are enforced during a depth-first traversal of the AST that occurs as a pass after traditional Java typechecking has occurred. This design allows JavaCOP rules to make use of Java type information, which is critical for many kinds of type extensions. 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;

Tree subclass	Name	Java example
`JCMethodInvocation`	Method call	`meth(args)`
`JCAssign`	Assignment	`x=y`
`JCClassDecl`	Class definition	`class X{ .. }`
`JCIdent`	Identifier	`foo`
`JCIf`	Conditional	`if(cond).. [else ..]`
`JCMethodDecl`	Method definition	`void foo(){..}`
`JCNewClass`	Instance creation	`new World("hello")`
`JCReturn`	Return statement	`return false;`
`JCFieldAccess`	Field Selection	`s.field`
`JCSkip`	Empty statement	`;`
`JCTypeCast`	Cast	`(String)s`
`JCVariableDecl`	Variable declaration	`String s = "hello";`

Fig. 2. A selection of AST nodes classes and their meanings.

Java interfaces are represented by class types internally. Each AST node representing a declaration also maintains its set of Java metadata annotations [Bloch 2002], which can be used as type annotations by programmers and then leveraged in JavaCOP rules.

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. Given `javac`'s modular compilation, the source of some depended-upon program entities and the AST nodes for those entities, may be unavailable. The `javac` compiler reconstructs necessary information about nonlocal entities from their bytecode representations, and stores it as `Symbol` objects.

2.2 AST Rules

As shown in Listing 1, a JavaCOP rule is a function that starts with the keyword `rule` and includes a name, a "joinpoint" that determines when a rule is applicable, and a body containing a sequence of constraints. An *AST* joinpoint consists of a single formal parameter whose type is a (subtype of) `JCTree`. 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 Listing 1 will be passed each node representing a Java assignment statement during JavaCOP's traversal of an AST. The rule employs two user-defined helper predicates (described later in this section) to require that the right-hand-side expression in an assignment be definitely nonnull whenever the type of the left-hand-side variable or field is declared (via a metadata annotation `@NonNull`) to be nonnull.

In a traditional type system, a program is considered to typecheck successfully if there is some way to derive a type for the program through the given typechecking rules. Because pluggable type systems in JavaCOP often impose only a few additional constraints onto the existing Java type system, we use the opposite convention. In particular, a program (or compilation unit) successfully typechecks by default in JavaCOP, and JavaCOP rules are used to impose

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

Listing 1. A JAVACOP rule which prevents assignment of a possibly null value into a `@NonNull` reference.

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

Listing 2. A rule which enforces proper subtyping constraints for a *nonnull* type system.

additional requirements to be satisfied. However, it is easy for a JAVACOP user to implement the traditional style if desired; an example of this style is shown in Section 2.5.

2.3 Subtype Rules

A hallmark of most object-oriented type systems is the notion of subtyping, which safely allows values of one type to be viewed as having a different type. For example, the type of the right-hand side in an assignment statement may be a subtype of the type of the location being assigned, and the actual arguments to a method call may be subtypes of the corresponding formal argument types. Pluggable type systems for object-oriented languages may need 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, the rule in Listing 2 subsumes the `checkNonNull` rule shown previously. The only syntactic difference is the joinpoint of the form `node <<: sym`. The new rule applies to any AST node where a subtype relationship is traditionally required, including assignment and return statements, parameter passing, and type casts. Here, `node` is the expression which is being "viewed" at the type of the identifier represented by the symbol `sym`. At an assignment node, for example, the right-hand-side expression will be checked against the symbol for the variable or field being assigned into.

2.4 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 boolean condition evaluates to `true`; otherwise the constraint fails. As a simple example, the following rule shows how JavaCOP can encode the semantics of the `final` modifier for Java classes.

```
rule finalClass(JCClassDecl c){
  require(!c.supertype.sym.hasAnnotation("Final")):
    error(c, "Superclass has a @Final superclass!");
}
```

This rule checks each class definition to ensure that the class does not inherit from a class that has the `@Final` attribute. The constraint in the rule employs the rule language's `Symbol` objects to access *interface* information about a class's declared superclass.

Often a constraint should only be applied under certain circumstances. This can be accomplished through the use of a `where` constraint. Like `require`, a `where` constraint takes a boolean expression as its condition. In addition, a `where` constraint has 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 Listing 2. In that rule, an expression only needs to be shown to be nonnull if it is flowing into a variable or field that is annotated with `@NonNull`.

The language for constraint conditions is an extension of the language for Java `boolean`-typed expressions. These expressions can invoke methods on any AST nodes, types, and symbols in the scope of the constraint. JavaCOP also supports conditions that perform let-binding type tests and structural pattern matching, which are discussed in Section 2.6. Constraints 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.

2.5 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's `override` [Igarashi et al. 2001]), 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 2.3 makes use of a `requiresNonNull` predicate which is defined next.

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

The `requireNonNull` 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 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 augment or modify the annotation mechanism. For example, nonnullness could be indicated by using a marker interface instead of Java's metadata facility, simply by changing the definition of the `requiresNonNull` predicate.

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. 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 Listing 2 makes use of the `definitelyNotNull` predicate, which checks whether a given `JCTree` object is definitely nonnull. In JavaCOP , we can define this predicate with a case analysis on different subtypes of `JCTree`, by providing multiple definitions of the predicate. A few representative definitions are provided in Listing 3. 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 third definition in Listing 3 fails if the given node does not represent a Java `new` expression.

As mentioned in Section 2.2, predicates also allow developers to define type systems in a more traditional fashion where every type-correct expression must be defined explicitly, as opposed to the allowed-by-default semantics of JavaCOP rules. For instance, the developer could define a subtyping predicate for nonnull that must hold at all places where subtyping is required. Instead of defining the subtyping rule as defined in Listing 2, we could instead define the rule and a predicate as shown in Listing 4. Unless one of the `isSubtype` predicate definitions is satisfied, this rule will fail.

2.6 Pattern Matching and Conditional Assignment

Type rules often depend on the ability to deconstruct the expressions, types, and environments that they constrain, so it is imperative that a language

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


/* The value of an assignment is non-null
   if the value being assigned is non-null */
declare definitelyNotNull(JCAssign a){
  require(definitelyNotNull(a.rhs));
}


/* Class instantiation results are never null */
declare definitelyNotNull(JCNewClass n){
  require(true);
}
```

Listing 3. Three definitions of a predicate which indicates whether an expression definitely evaluates to a nonnull value.

```
rule checkNonNull(node <<: sym){
  require(isSubtype(node, sym))
    :error(node, "Non-null subtyping constraints violated");
}


declare isSubtype(JCTree node, Symbol sym){
  require(!requiresNonNull(sym));
}


declare isSubtype(JCTree node, Symbol sym){
  require(requiresNonNull(sym) && definitelyNotNull(node));
}
```

Listing 4. A replacement rule for `checkNonNull` from Listing 2 which employs a more traditional subtyping relation via the `isSubtype` predicate.

designed for encoding pluggable types also have this ability. To that end, JavaCOP supplies two new operators: `<-` for type-conditional value binding, and `=>` for pattern matching on AST nodes.

An expression of the form `v <- e` evaluates to `true` if `e` is found to be an instance of the declared type of the variable `v`. If satisfied, `e` is cast to the type of `v`, which is then let-bound to this value. Otherwise, the type-conditional binding evaluates to `false` and the value is not bound. A constraint condition may be preceded by a list of variable declarations to be bound within its conditional expression. For example, the following constraint is satisfied only for field accesses of the form `x.f`, where `x` is a simple variable. In that case, the rule binds (the AST node for) `x` and `f` to new names `recv` and `fname` for use in the rest of the rule.

```
require(JCIdent recv, String fname; recv <- dref.selected
                          && fname <- dref.name){...}
```

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. A detailed description of the syntax of JavaCOP patterns is discussed elsewhere [Markstrum 2009]. Pattern matching can significantly improve the readability of JavaCOP rules. As a simple example, the following pattern match succeeds if `decl` is a method declaration that is annotated with the `@NonNull` attribute; it additionally binds the method's result type and name for later use.

```
where(Tree typ, String name; decl => [@NonNull typ name(...)])
```

The pattern `...` matches against any number of elements in a sequence, thereby allowing the method to have any number of formal parameters.

2.7 Quantification

JavaCOP provides quantification over two kinds of data structures. First, constraints may universally or existentially quantify over `javac` `List`s with `forall` and `exists` quantifiers, respectively. The syntax is similar to the syntax of the enhanced for loop in Java. For example, the predicate defined shortly requires that every JUnit `@Test` annotation on a particular symbol defines its `expected` attribute. The `forall` iterates over a list of all annotations found on the `Symbol`, binding each to the name `c` in turn. The syntax for existential quantification is analogous.

```
declare hasExpectedMetadata(Symbol s){
  forall(Compound c : s.attributes){
    where(c.fullName.equals("@org.junit.Test")){
```

```
      require(c.getComponent("expected") != null);
}}}
```

Second, JavaCOP allows quantification 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 quantified variable are considered. For example, the predicate defined shortly determines if a method defined in the `org.junit.Assert` class is called at least once within the body of the method `m`. The defining class is obtained by looking up the `owner` of the method. The rule uses existential quantification to iterate over every method invocation (`JCMethodInvocation`) in the body of the given method declaration (`m`).

```
declare callsAssertMethod(JCMethodDecl m){
  exists(JCMethodInvocation a : m){
    require(a.meth.owner.fullName.equals("org.junit.Assert"));
}}
```

2.8 Error Reporting

As with any implementation of a type system, it is important to provide programmers with 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* on constraints, 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`; the AST node at whose source position the error is to be reported; 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 [2004] might treat all violations as warnings, while a use of pluggable types to enforce a form of security might treat violations as fatal errors. When a constraint fails to be satisfied, JavaCOP searches for the nearest enclosing failure clause and executes it.

2.9 Escaping to Java

While the JavaCOP rule language is expressive enough for most purposes, some developers may desire the expressiveness of the full Java language. Therefore we provide a simple "escape hatch" that allows JavaCOP rules to interact with arbitrary Java code: regular Java fields and methods may be invoked on an object by using the `#` operator in place of the usual `.` operator. As we describe in Section 4, this simple mechanism allows JavaCOP rules to easily access the results of user-defined dataflow analyses in order to incorporate flow-sensitive reasoning.

3. CASE STUDIES: DOMAIN-SPECIFIC CHECKERS

This section illustrates the practicality of the JAVACOP rule language through two case studies that enforce domain-specific requirements for existing Java libraries. These examples are a particularly compelling application for pluggable type systems, because it would never make sense for such requirements to be enforced as part of the type system of a general-purpose programming language like Java. JAVACOP allows programmers to precisely document domain-specific requirements in a declarative manner and has automatically identified dozens of errors in widely used Java code.

3.1 Design Patterns in Polyglot

Polyglot [Nystrom et al. 2003] is an extensible compiler framework for Java from Cornell, written in Java. Polyglot has been publicly available since 2004 and used by many researchers to implement Java language extensions [POLYGLOT 2004]. Polyglot employs a number of design patterns that are not checked by the standard Java type system, so programmers must manually ensure that their code conforms. We implemented a pluggable type system for Polyglot in JAVACOP to automatically check for proper adherence to the following idioms.

(1) Polyglot employs the factory design pattern [Gamma et al. 1995] for both AST nodes and for "type objects" that hold the type information about a class. Our checker requires that any expression of the form `new C(...)`, where `C` is a subtype of the `Node` interface, appear only in classes that subtype `NodeFactory`, and similarly for type objects.
(2) Each AST node in Polyglot is represented by both a class and an interface. The intent is that clients of a Polyglot extension should only manipulate AST nodes through their associated interfaces. Our checker requires that a node class is never used as the type of a public or package-level field or as the argument or result type of a public or package-level method.
(3) Polyglot uses a variant of the visitor design pattern [Gamma et al. 1995] to allow implementers to traverse the AST. Each node class must have a `visitChildren` method that implements the traversal behavior for that kind of node. Our checker requires that each node class overrides the `visitChildren` method if it adds at least one new field of type `Node` (or a subtype).
(4) Polyglot employs a notion of *delegates* [Nystrom et al. 2003] that allows the behavior of an AST node to be modified modularly without requiring the creation of a subclass. Each AST node has a pointer to a delegate object, and clients should always invoke certain operations of a node (defined in the `NodeOps` interface) through the node's delegate (e.g., `n.del().typecheck()` instead of simply `n.typecheck()`). Our checker enforces this rule.

The entire Polyglot checker consists of 80 lines of (nonblank, noncomment) JAVACOP rules and auxiliary predicates. JAVACOP'S declarative nature makes each rule relatively straightforward to understand. For example, Listing 5

Table I. Results of Running the Polyglot Style Checker

		Errors	
Compiler	*LOC*	*signaled*	*actual*
Polyglot-1.3.4	20910	7	7
Polyglot-2.3.0	25154	7	7
Polyglot5	7800	3	3
JPred	3343	0	0
eJava	2458	2	2
jet-0.9.0	921	0	0
jif-3.1.1	22020	14	12

```
rule nodeFactory(JCNewClass nc) {
  where(isSubtype(nc.constructor.owner, "polyglot.ast.Node")) {
    require(isSubtype(env.enclClass.sym, "polyglot.ast.NodeFactory")):
      error(nc, "Nodes cannot be directly instantiated "+
                "outside of the node factory!");
}}
```

Listing 5. A JAVACOP rule enforcing Polyglot's factory design pattern.

enforces the factory design pattern for AST nodes. The rule directly corresponds to the English description provided earlier. The auxiliary user-defined predicate `isSubtype` checks whether a type (represented by its symbol) has a particular supertype.

We ran our Polyglot style checker on seven Polyglot compilers or extensions; the results are shown in Table I. The first two compilers are the Polyglot 1.x and 2.x branches from Cornell, respectively. The next three compilers in the table are Polyglot extensions from UCLA to respectively support Java 1.5 features [POLYGLOT5 2007], predicate dispatch [Millstein et al. 2009], and expanders [Warth et al. 2006]. The last two compilers are Polyglot extensions from Cornell to respectively support nested intersection [Nystrom et al. 2006] and secure information flow [Myers 1999].

The second column in the table lists the (nonblank, noncomment) lines of Java code in the compiler or extension, ignoring generated code (e.g., from the parser generator) and other special files. For each compiler, we list the number of errors signaled by the checker. We provided these error messages to the developers of the compilers and asked them to verify which were actual errors. As the last column in the table shows, of the 33 errors signaled across all compilers, 31 of them were actual errors.

All 14 errors in the two Polyglot base compilers were violations of the fourth idiom described previously, related to delegates. These 14 errors represent 12 distinct errors: two errors from the 1.3.4 version were duplicated in the 2.3.0 version. All of these errors have now been fixed by the developer.

Nine of the 14 errors signaled for Jif were violations of the first idiom described earlier, related to factories. Two of these nine errors were considered false positives by the developer. One involved a temporary node class used only

during parsing, which was deliberately not given an associated factory method and so is always directly instantiated. The other pertained to a class that was being used as a factory for certain type objects, even though it was not a subtype of the standard Polyglot interface for type factories. A user could easily employ an annotation like `@TypeFactory`, along with a simple modification to our JAVACOP rule, to eliminate this false positive.

3.2 JUnit Checker

JUnit [JUNIT 2000] is a popular unit testing framework for Java, with widespread use in both academia and industry and special support in Integrated Development Environments (IDEs) such as Eclipse. The focus of this section is JUnit 3, which uses pre-Java 5.0 features, as opposed to the more recent JUnit 4, due to the earlier version's continued prominence. We have, however, implemented style checkers for both JUnit 3 and JUnit 4.

In order to create a test to be run by JUnit, a programmer defines a subclass of `junit.framework.TestCase`. This class should contain one or more test methods whose names begin with the string `"test"`. When running the tests, the JUnit framework only runs one test method per instance. Which method to run is indicated to the framework by calling `super(methodName)` from the constructor of the `TestCase` subclass. The test methods will most likely need to invoke methods on objects from the application being tested. These objects should be constructed by overriding the `setUp` method of `TestCase` and should be cleaned up by overriding the `tearDown` method. Once a test method has made calls on the application's objects, it should ensure that the calculation has the expected result and indicate to the JUnit framework whether or not the test succeeded or failed. This is done by calling one of the static methods from `junit.framework.Assert` (e.g., `assertEquals(expected, actual)`.

We have built a checker which helps to find errors in JUnit 3 test cases that might cause tests not to be run or failures not to be reported. This checker has 59 lines of code, consisting of 10 predicate declarations and one rule.[1] The checker enforces the following discipline.

(1) Constructors for classes which extend `junit.framework.TestCase` must include a call to `super(arg)` where `arg` has type `String`.
(2) If a test case overrides a previously redefined `setUp` or `tearDown` method, then it should include a call to the superclass method. This ensures proper initialization and clean up of inherited state and correct behavior of tests in the superclass.
(3) Test methods must include a call to an assertion method from the class `junit.framework.Assert`.

We ran our JUnit style checker on the test suites of three open-source Java projects, the results of which are shown in Table II. The first project, Korat, is a framework for automated testing of Java programs [Boyapati et al. 2002;

[1]For JUnit 4, our checker has 39 lines of code, eight predicate declarations, and two rules in its body.

Table II. Results of Running the JUnit Style Checker.

Program	*# Test Classes*	*# Test Methods*	*Conforming Classes*	*Conforming Methods*	*# Annotations*
Korat-1.0	15	45	3	15	0
w/ annotations			14	45	6
WALA-1.0.0	29	137	13	41	0
w/ annotations			20	122	15
X10-1.0.70	8	317	6	28	0
w/ annotations			6	316	2

Milicevic et al. 2007]. WALA [WALA 2007] is a static analysis framework from IBM. For WALA, we look at only the tests from the `com.ibm.wala.core.tests` package. The third project is IBM's concurrent language X10 [Charles et al. 2005].

The second and third columns of Table 3.2 present the number of test classes defined in the test suites and the total number of test methods. Most of the test classes define multiple test methods, but it is worth noting that for X10, one of the classes defines 289 test methods. The fourth and fifth columns of the table indicate the number of classes and methods that conform to all properties. From the initial results of running our checker (the top number in the fourth and fifth columns), it appeared that these test suites generally do not conform to recommended style guidelines; specifically many test methods did not include a call to a JUnit assertion method.

A review of these style errors revealed that many of the test methods were depending on helper methods to perform a JUnit assertion test. In order to allow our checker to recognize and accept this pattern, we created an annotation and placed it on these helper methods. Test methods that invoked a method with this annotation were then treated as if they had performed an assertion test. The annotated methods were also checked to ensure that they did in fact do an assertion test. The modified rule set contains 67 lines of code, consisting of 12 declarations and one rule. One of the two new declarations is shown in Listing 6, which checks whether a JUnit assertion method or a method with the `@JUnitAssert` annotation is called in the body of a method declaration.

The results of including the annotations are also found in Table 3.2. With a minimal number of annotations, nearly all of the test methods were found to invoke some JUnit assertion test. However, several violations of our checker remain. First, there are still 16 methods that never invoke a JUnit assertion method. Three of these are actual errors; the test cases can never fail and so are useless. The other 13, from WALA's test suite, invoke a regular Java `assert` rather than a JUnit assertion. While not in conformance with the JUnit guidelines on how to return test results, we could easily update our checker to allow this idiom. Second, there are four helper methods that themselves invoke auxiliary methods in order to call a JUnit assertion. We could capture this idiom through further annotations. Finally, two class constructors in Korat do not pass the name of the test method to the superclass constructor. Instead

```
declare callsAssertMethodOrAnnot(JCMethodDecl mdecl){
  exists(JCMethodInvocation a : mdecl){
    require(a.meth.owner.fullName.equals("junit.framework.Assert") ||
            a.meth.hasAnnotation("junit.framework.JUnitAssert"));
}}
```

Listing 6. A predicate which determines whether a JUnit assertion method or a method with a `@JUnitAssert` annotation is called within the body of a method declaration from an extended JUnit 3 checker.

they perform the test directly in the constructor. While these tests will be run, JUnit does not anticipate assertions during object construction and so reports the errors as JUnit framework errors rather than test failures.

4. THE JAVACOP DATAFLOW FRAMEWORK

Some programming disciplines require flow-sensitive reasoning. For example, a programmer should never dereference an object unless it can be guaranteed *at that program point* that the object is nonnull. A programmer might accomplish this by placing an if-statement checking for nullity around the dereference. Listing 7 illustrates code that a practical non-null checker should allow. The need for flow-sensitive reasoning in building expressive type systems has been demonstrated in recent research (e.g., Fähndrich and Leino [2003], Aldrich et al. [2002], Boyland [2001]).

We considered augmenting the JavaCOP rule language to directly support flow-sensitive reasoning. However, flow analyses are significantly different from traditional type-system constraints, for example, operating over a Control-Flow Graph (CFG) rather than an AST and requiring iteration until reaching a fixpoint. Furthermore, even when type systems do require flow sensitivity, such reasoning is typically only employed in limited ways for extra precision in special situations.

Therefore, we decided to retain the simplicity of our flow-insensitive rule language but provide a separate mechanism for incorporating flow sensitivity when desired. We provide an API in Java that allows users to easily define dataflow analyses that track user-defined dataflow facts. Such analyses are performed just after regular Java typechecking, before any JavaCOP rules are enforced. Results of the analyses are made available as decorations on the AST nodes, which allows these results to be easily accessed from the flow-insensitive JavaCOP rules.

While many frameworks for performing dataflow analysis on Java programs exist (e.g., Soot [Vallée-Rai et al. 1999]), none directly fits the needs of a pluggable type system. For instance, most dataflow frameworks work over an intermediate format such as three-address code. However, it is important for analyses in our context to operate over a representation of the source program so the results can be easily used when typechecking that program. While this constraint makes our framework's implementation more complex, we were able to hide this complexity from users, as described shortly. Also, as shown before for a nonnull type system, pluggable type systems often require a form of

```
class Person {
  @NonNull String name = "Chris";


  void setName(String newname) {
    if (newname == null)
      name = "";
    else
      name = newname;
  }
}
```

Listing 7. Example code that requires flow-sensitive reasoning.

path-sensitive reasoning, so our framework builds in a simple but useful mechanism for this purpose.

4.1 Specifying Dataflow Analyses

JAVACOP 's flow framework allows users to easily create forward, gen/kill style, intraprocedural dataflow analyses. In general, describing an analysis of this type requires specifying:

(1) the dataflow facts being tracked (e.g., definition sites for a reaching definitions analysis or program expressions for an available expressions analysis);
(2) the sets of dataflow facts that are generated by program expressions;
(3) the sets of dataflow facts that are killed by program expressions;
(4) how to combine sets of dataflow facts at control flow merges.

To define an analysis in our framework, the user provides a Java class that represents a set of dataflow facts and that implements the `FlowFacts` interface shown in Listing 8. The user can choose any representation for sets of dataflow facts that is appropriate for the analysis (e.g., a bit vector versus a hash map). The user-defined `genSet` and `killSet` functions determine what dataflow facts to propagate as a result of traversing the given expression. The `addSet` and `removeSet` functions, respectively, are used to update the current set of facts with the gen and kill sets, and the `meetWith` function is used to combine sets at control-flow merges.

Having a single gen and kill set for each expression, however, is not sufficient to define the analysis we need for our motivating example in Listing 7. The key expression, `newname == null`, is on the control-flow path to both the then branch and the else branch. In order to differentiate between these two cases, we allow for a limited form of path sensitivity in the interface: `genSetTrue`, `genSetFalse`, `killSetTrue`, and `killSetFalse`. Our framework uses these functions, rather than `genSet` and `killSet`, for boolean-typed expressions. This "branch-condition sensitivity" allows users to define different gen and kill sets based on whether a boolean expression evaluates to true or false. If a boolean expression is used in a context that does not affect control flow, the framework merges the two gen

```
interface FlowFacts {
  FlowFacts genSet(JCTree node);
  FlowFacts killSet(JCTree node);

  //Allow branch-condition-sensitive analysis
  FlowFacts genSetTrue(JCTree node);
  FlowFacts killSetTrue(JCTree node);
  FlowFacts genSetFalse(JCTree node);
  FlowFacts killSetFalse(JCTree node);

  //Operations for combining sets of FlowFacts
  FlowFacts addSet(FlowFacts f);
  FlowFacts removeSet(FlowFacts f);
  FlowFacts meetWith(FlowFacts f);
}
```

Listing 8. The interface for defining dataflow analyses.

sets (and the two kill sets) using the interface's `meetWith` method to determine the outgoing set of dataflow facts.

The analysis is specified on the AST of the source program, which is the same data structure manipulated in the JAVACOP rule language, thereby simplifying matters for users. Our dataflow framework automatically handles the details of properly traversing this AST. For example, a complicated expression is visited one subexpression at a time, in control-flow order, invoking the user-defined `genSet` and `killSet` functions to propagate dataflow facts among these subexpressions. Similarly, the framework handles all control flow automatically, including complex control flow arising from `try/catch/finally`, labeled breaks and continues, etc., invoking the user-defined `meetWith` function at each merge point.

4.2 An Example Analysis for the Nonnull Checker

Listing 9 shows the `FlowFacts` implementation for an analysis that can determine that `newname` is nonnull when assigned to `name` in Listing 7. The `genSetFalse` implementation checks if the given expression has the form `"localVar == null"` . If so, it returns a set containing the fact that `localVar` is nonnull, since this clearly holds when the expression evaluates to false. This fact might be invalidated by a reassignment to `localVar`. This is handled by the `killSet` implementation. The implementations for the other gen and kill functions required by the `FlowFacts` interface are not shown and simply return an empty set of dataflow facts. Since the framework handles all Java control-flow statements, this implementation suffices to determine all of the dataflow facts shown in the comments of Listing 10. Our actual implementation enhances this one by allowing `null` to appear on the left of an equality test and allowing for `!=` operators. Our implementation additionally generates nonnull facts on certain assignments to local variables that are clearly not null, such as string literals and (boxed) primitive literals.

```
class NonNullFacts implements FlowFacts {

  HashSet<Symbol> nonnulls = new HashSet();

  // Explicit non-null test on local variable generates non-null fact
  FlowFacts genSetFalse(JCTree node){
    NonNullFacts gen = new NonNullFacts();
    if (node instanceof JCBinary && node.tag == JCTree.EQ) {
      JCBinary b = (JCBinary)node;
      Symbol s = getSymbol(b.lhs);
      if (b.rhs.getKind() == Kind.NULL_LITERAL && isLocal(s))
        gen.nonnulls.add(s);
    }
    return gen;
  }

  //Re-assignment kills non-null fact for local variable
  FlowFacts killSet(JCTree node){
    NonNullFacts kill = new NonNullFacts();
    if (node instanceof JCAssign){
      Symbol s = getSymbol(((JCAssign)node).lhs);
      if (isLocal(s))
        kill.nonnulls.add(s);
    }
    return kill;
  }

  //Define the meet operation as intersection
  FlowFacts meetWith(FlowFacts f){
    nonnulls.retainAll(((NonNullFacts)f).nonnulls);
    return this;
  }

  //Javacop rules can use this method to query non-null facts\\
  boolean isIdentNonnull(JCIdent id){
    return nonnulls.contains(getSymbol(id));
  }

  // code to implement remaining interface methods
   ⋮
}
```

Listing 9. FlowFacts for nonnull analysis.

```
String s = null;
while (s == null) {
  if (...) s = "done";
}
// s is non-null here
s = null;
while (s == null) {
  try {
    if (...) s = "done";
    if (...) throw new Exception();
  } catch(Exception e) { break; }
}
// s might be null here!

boolean empty = (s == null) || /* s non-null here */ s.length() == 0;
```

Listing 10. Some examples of dataflow facts determined by the *branch-condition-sensitive* NonNullFacts analysis in the face of complex control flow.

```
flowfact nonnull.NonNullFacts;

declare definitelyNotNull(JCIdent id){
  require(NonNullFacts f; f <- id.getFlowFacts("nonnull.NonNullFacts")){
    require(f#isIdentNonnull(id));
}}
```

Listing 11. A JavaCOP declaration using nonnull dataflow facts.

4.3 Accessing Analysis Results from JavaCOP Rules

After a dataflow analysis is run, each expression node in the AST is decorated with the set of dataflow facts valid before its evaluation. The JavaCOP rules can then simply query these decorations (using a new `getFlowFacts` function) in order to incorporate flow-sensitive reasoning. Listing 11 shows the JavaCOP code that we must add to the nonnull checker in order for JavaCOP to accept the code in Listing 7. The `flowfact` declaration allows the user to specify that a particular implementation of the `FlowFacts` interface should be used during the analysis pass. A JavaCOP file may indicate that several analyses should be performed by including multiple `flowfact` declarations.

Incorporating flow sensitivity into our nonnull checker only requires adding one new case to the `definitelyNotNull` predicate from Listing 3, as shown in Listing 11. The case accesses the `NonNullFacts` object that decorates an identifier's AST node in order to check whether or not the dataflow analysis determined the identifier to be nonnull at this point. Note that the `FlowFacts` interface itself does not provide any special means of querying the dataflow facts. The user is free to provide whatever methods are appropriate and convenient for querying a particular implementation. In this example, the `isIdentNonnull` method from Listing 9 serves this purpose.[2]

[2]Recall that the # operator allows access to Java methods that are not part of the JavaCOP API.

4.4 Implementation

As mentioned earlier, our flow analysis must work over a representation of the source program rather than an intermediate representation. Fortunately, the existing Java typechecker also has this problem: it must perform a flow analysis on a source program in order to flag errors for uninitialized local variables and final fields, unreachable code, and uncaught exceptions. We decided to adapt the well-tested and efficient code for that flow analysis in the `javac` compiler for our purposes by generalizing it to make calls to our `FlowFacts` interface during its traversal of the AST. This decision significantly simplified our implementation, since the original code was already properly handling the complexities of compound expressions in Java as well as control flow via exceptions, `finally` blocks, short-circuited boolean logic, etc.

5. CASE STUDIES: ADVANCED TYPE SYSTEMS

This section demonstrates JavaCOP's ability to express state-of-the-art type systems that require disparate forms of flow-sensitive reasoning and presents our experience implementing and using them. First we complete the description of our example checker for preventing null pointer dereferences in Section 5.1. Then we present a system for ensuring uniqueness of pointers in order to control aliasing in Section 5.2. Recent research (e.g., Fähndrich and Leino [2003], Aldrich et al. [2002], Boyland [2001]) has shown that type systems for these properties that are both sound and expressive enough to be usable in practice employ reasoning that is subtle and complex. By using JavaCOP's declarative rule language introduced in Section 2 in concert with the dataflow framework presented in Section 4, we are able to build robust, practical checkers for these properties.

5.1 Nonnull Type System

Adding raw types. The examples throughout this article have established the basic features of our nonnull checker. Fields or variables with explicit `@NonNull` annotations are guaranteed to contain nonnull values. Using the JavaCOP dataflow framework, explicit runtime checks in the source code can be used to ensure nonnullity of a reference before dereferencing it or storing it into a location annotated with `@NonNull`. Unfortunately, due to the semantics of Java object construction, it is impossible to guarantee that a field annotated with `@NonNull` *never* contains `null`, even if it is initialized at the declaration site. The code in Listing 12 shows how a field can be accessed before it is initialized. Since `A`'s constructor is executed before subclass `B`'s field initializers are run, the overridden `init` method in `B` will dereference field `f` before it is initialized. The Java runtime will have stored `null` in `f` and a null pointer exception will result.

As this example shows, it would not be sound for our checker to simply enforce that a field annotated with `@NonNull` have a nonnull initializer in its declaration. Instead, our checker supports the *raw types* approach invented by others [Fähndrich and Leino 2003]. We assume that fields declared as nonnull may in fact be null while the object is under construction, or *raw*. We introduce

```
class A {
  A() {
    init();
  }

  void init() { }
}

class B extends A {
  @NonNull String f = "not␣null";

  void init() {
    // Executes before f is initialized
    System.out.println( f.length() ); //null deref!
  }
}
```

Listing 12. Java fields may be accessed before their initializer is run.

```
class A {
  A() {
    init();
  }

  @RawThis void init() { }
}

class B extends A {
  @NonNull String f = "not␣null";

  @RawThis void init() {
    if (f != null)
      System.out.println( f.length() );
  }
}
```

Listing 13. Using raw types to guard against null dereferences during object construction.

Table III. `@NonNull` Annotation Results for Dijkstra's Algorithm Implementation and JAVACOP

	Dijkstra	*JavaCOP*
LOC	629	948
(add'l code dependency)		∼1000
Derefs	206	628
`@NonNull` annotations	83	92
(add'l code dependency)	43	100
`@RawThis` annotations	1	0
(add'l code dependency)	1	1
Nullity checks	46	93
bugs	7	7
false positives	0	22
unknown	0	14
Java limitations	39	50

Additional code dependencies listed here include wrapper methods for library calls as well as additional annotated code. Nullity checks inserted due to lack of Java support for annotations on enhanced for loops and generics are listed under Java limitations.

two new annotations, `@Raw` and `@RawThis`, that indicate that a method parameter or the receiver of a method call, respectively, may be under construction. JAVACOP rules are used to enforce that a constructor only invokes `@RawThis` methods on the object being constructed and only passes `this` to a method as a `@Raw` parameter. Listing 13 fixes the erroneous code from Listing 12 so that our checker accepts it. The type system requires `init` to have the `@RawThis` annotation, since it is called while the object is under construction. Once the method is marked as having a potentially raw receiver, the checker requires that a runtime nullity check is inserted before dereferencing field `f`, since it may contain `null` despite its `@NonNull` annotation.

Once the raw types mechanism is in place, it is overly restrictive to insist that a `@NonNull` field be initialized at its declaration site. It is sufficient to check that all `@NonNull` fields have been assigned a nonnull value by the end of construction. This check requires dataflow information, since it must reason about all paths through the constructors. Although the need for such a definite assignment analysis was not anticipated when we designed our dataflow analysis framework, it is easily supported by that framework.

The entire nonnull checker, including raw types, required 136 lines of JAVACOP code consisting of 7 predicates and 12 rules. The two flow analyses used by the JAVACOP rules were built using our dataflow framework in a total of 147 lines of Java code.

Experience using the nonnull checker. In Table III we present results of applying our nonnull checker to two existing Java programs to make them safe from null dereferences. The first column contains results pertaining to an undergraduate project by one of the authors that uses Dijkstra's algorithm for determining shortest path on a given street map. The second column contains results from applying JAVACOP to itself, namely the pass that we added to the OpenJDK compiler for JAVACOP'S rule enforcement. The table lists the size of

each application and the number of object dereferences that the nonnull type system must prove safe.

The table also lists the number of annotations required, both within the application itself and within depended-upon code: the Java standard library and (for JavaCOP) the rest of the `javac` compiler implementation. The number of annotations could be significantly reduced through the use of appropriate defaults [Chalin and James 2007]. Such a nonnull-by-default type system could be adapted from the nonnull type system discussed here by changing the predicate `requiresNonNull` shown in Section 2.5, which defines the nonnull type, to require the lack of a `@Nullable` annotation instead of the presence of a `@NonNull` annotation.

The "Nullity checks" category indicates the number of places in which we had to add an explicit test for nonnullness in order to typecheck successfully. These checks have been partitioned into several categories. The seven bugs in the Dijkstra application all have to do with improper handling of input files. If the files are not in the correct format, the implementation generates null pointers within its data structures, which can later be dereferenced. Seven bugs were also found in the JavaCOP source. For example, the expression `filename.getParentFile().toURL()` contains an error, since the method `getParentFile()` may return `null` for a malformed file descriptor. We list 14 nullity checks as "unknown". These checks all pertain to references to `javac Scope` objects. Our inspection of the `javac` code leads us to believe that the `Scope` objects are phased: initialized to `null` but, at some time before the JavaCOP pass, set to a nonnull value. However, we found no conclusive evidence to support this belief and so left the checks uncategorized.

The code in Figure 3(b) illustrates an example false positive. The type system complains that the potentially null field `tail` is being assigned to the `@NonNull` variable list. However, the loop guard ensures that `tail` is nonnull, as shown in Figure 3(a). We satisfy the type system by inserting a nullity check, as shown in Figure 3(c). This code also illustrates the need to introduce local variables, since the type system only supports flow sensitivity for local variables. Flow sensitivity for fields is more challenging due to the potential for aliasing and the possibility of concurrent access by multiple threads.

Finally, "Java limitations" lists nullity checks due to limitations in Java's annotation syntax. Most notably, type parameters cannot have annotations, so for example it is not possible to have a `List` of `@NonNull` strings. Therefore each time we access and use an element from such a list, a spurious nullity check is required. This limitation has been recently resolved in Java 7 [Ernst 2007].

5.2 Unique Type System

We have also built a uniqueness checker in JavaCOP which incorporates two important features that have been described in recent literature: lent parameters and reassignment of unique objects without destructive reads [Aldrich et al. 2002; Boyland 2001]. These features are key to building a practical uniqueness checker for Java. Our experience building this checker illustrates the expressiveness of JavaCOP , and of the flow analysis framework in particular.

(a)

```
public class JCList<T extends Object> {
                    ⋮
  public boolean nonEmpty(){
    return tail != null;
  }
}
```

(b)

```
@NonNull JCList<String> list = ...;
for ( ; list.nonEmpty(); list = list.tail) {
  /* loop body */
}
```

(c)

```
@NonNull JCList<String> list = ...;
for ( ; list.nonEmpty(); ) {
  /* loop body */
  JCList<String> tail = list.tail;
  if (tail == null)
    throw new RuntimeException(...);
  list = tail;
}
```

Fig. 3. Example code (b) demonstrating a false positive indicated by our `@NonNull` type system and how the code had to be modified (c) to appease the type system. This is a false positive because the type system is unaware of the invariant which the method `nonEmpty()` (a) ensures.

The goal of the uniqueness checker is to enforce that an object stored in a variable or field annotated as `@Unique` has no aliases: it is not directly accessible via any other field or variable. Toward this end, the JAVACOP rules for the checker limit the expressions that can be assigned into a `@Unique` location. For example, an unannotated field cannot be stored into a variable or field annotated as `@Unique`.

The checker must also prevent the creation of aliases to objects stored in a `@Unique` location. But maintaining a strict no-aliasing invariant is too restrictive to be practical. A programmer may want to pass a unique field to a utility method. For example, a list stored in a `@Unique` field may need to be passed to a sorting method, creating a local alias (the formal parameter name). The `@Lent` annotation for parameters supports this pattern by allowing for temporary aliasing of `@Unique` locations in the scope of a method call. The JAVACOP rules enforce that `@Lent` parameters are never stored into nonlent locations, thus preventing the creation of permanent aliases to objects in `@Unique` locations. The class at the top of Listing 14 shows example code that our uniqueness checker would reject. The class on the bottom fixes the errors. The first error shown in the figure demonstrates that in order to pass `@Unique` field `u` to the utility method `printObj`, the method's parameter must be annotated with `@Lent`. Once the `@Lent` annotation is in place, the checker forbids the method from storing the object. We also support a `@LentThis` annotation which indicates that the implicit `this` parameter of an instance method should be treated as lent.

```
class ClassWithErrors {
  @Unique Object u;
  Object o;


  void m() {
    u = new Object();
    printObj(u); // error! can't pass unique to non-lent
    o = u; // error! creates alias to object in u
  }


  void printObj(Object obj) {
    o = obj;
    System.out.println(o.toString());
  }
}

class ClassWithoutErrors {
  @Unique Object u;
  Object o;


  void m() {
    u = new Object();
    printObj(u); // ok, since param is lent
    o = u; // ok, since u "dead" here
    u = new Object();
  }


  void printObj(@Lent Object lentobj) {
    //Can't store ref to lent object
    System.out.println(lentobj.toString());
  }
}
```

Listing 14. Example code using uniqueness annotations to control aliasing.

Table IV. Two Approaches to Enforce that a Unique Location is Not Dereferenced while Alias Exists

Expression	*Backward Analysis*	*Forward Analysis*
read location x	generate "x is live"	enforce x $\notin$ must-be-dead set
method exit / method invocation	generate "f is live" for all fields	enforce no field f $\in$ must-be-dead set
write location x	kill "x is live"	kill "x must be dead"
assign from unique location u	enforce u $\notin$ live set	generate "u must be dead"

The second error in Listing 14 demonstrates another important feature of the uniqueness checker: we must be able to *remove* an object from a `@Unique` location and store it somewhere else. For this reason, we can't simply include a JAVACOP rule that forbids a `@Unique` location from appearing on the right-hand side of an assignment. Allowing the transfer of an object from a `@Unique` location to a new location in a language without a destructive read operation, like Java, requires that we slightly weaken our uniqueness invariant [Boyland 2001]. Rather than enforcing that a nonlent alias never exists, we maintain that a nonlent alias does not exist *at any time the* `@Unique` *location is read*. Enforcing this property requires a form of live variable analysis to ensure that a `@Unique` field or variable is dead (not read again before being reassigned) after it is stored into another location. For example, our checker signals the second error in method `m` of `ClassWithErrors` since we have to assume that field `u` is live after the method completes. As such, the object stored in `@Unique` field `u` may be accessed while an alias to the object exists, namely `o`. This error is fixed in `ClassWithoutErrors` by reassigning `u` to a new object before the end of the method. This makes `u` dead immediately after the assignment to `o` (assuming a single-threaded environment).[3]

Initially, it appeared that we would not be able to encode a uniqueness type system in JAVACOP since liveness is a backward analysis, and the JAVACOP dataflow framework only supports forward analyses. But we were able to devise a forward analysis that gathers the information needed by our uniqueness checker. Rather than generating liveness information, we generate "must be dead" facts whenever a `@Unique` location is stored into a nonlent location. The JAVACOP rules for our uniqueness checker can then flag an error whenever a location that should be dead is read or a must-be-dead fact for a field reaches a method invocation or the end of a method. The two approaches are summarized in Table IV. The forward approach was easily encoded as a `FlowFacts` implementation in our framework. We expect this general approach of translating checks that use backward dataflow information into different checks on forward dataflow information to be applicable to other problems.

We built the uniqueness checker using 150 nonblank, noncomment lines of JAVACOP code consisting of 14 predicates and 18 rules. The rules make use of the must-be-dead analysis described earlier which is a FlowFacts class with 122 lines of Java code.

6. TESTING PLUGGABLE TYPE SYSTEMS

Just as Java programmers make mistakes in their programs, JAVACOP users may introduce errors in their pluggable type systems. The standard way to ensure that a formal type system is correct in the research literature is to prove a *type soundness* theorem. However, it would be unreasonable to require JAVACOP users to manually prove a soundness theorem in order to gain confidence in their pluggable type systems. In a precursor project to JAVACOP , we explored

[3]To ensure soundness in a multithreaded environment, we could combine the uniqueness checker with a race condition checker and add a requirement that any two assignments which effect the "transfer" of a unique reference happen within the same `synchronized` block.

an approach to automatically discharging proof obligations to ensure soundness for user-defined type qualifiers, using an automatic theorem prover [Chin et al. 2005]. While this approach was sufficient for relatively simple type-system extensions, it does not scale to handle the kinds of sophisticated type systems shown in the previous section without significant manual intervention.

Therefore, in JAVACOP we have pursued an alternative approach that we believe is more practical: *testing for type soundness*. We provide a test harness that supports a novel two-stage testing process. The first stage provides the benefits of a traditional unit-testing framework, while the second stage helps JAVACOP users to directly identify violations of type soundness. We describe our approach next and discuss our experience using our test harness.

6.1 Two-Stage Testing Approach

The first stage, the *compilation* stage, of our test harness acts like a traditional unit-testing framework. A JAVACOP user can provide the pluggable type system being tested, a suite of Java programs to use as test cases, and the expected outcome of typechecking each test program with the given pluggable type system. Our test harness compiles each program in the test suite using the specified JAVACOP type system and reports violations of the expected outcomes. This stage helps to ensure that a pluggable type system continues to behave as expected as the type system evolves over time.

The second stage of our test harness, the *execution* stage, is motivated by the observation that pluggable type systems are often used to prevent a particular set of runtime errors. For example, in a nonnull type system, a field or variable declared `@NonNull` is intended to never have the value `null` at runtime, thereby preventing null dereferences. Such errors correspond to the "stuck expressions" which a standard type soundness theorem ensures can never be reached during execution of a well-typed program [Pierce 2002]. Rather than proving that such errors can never be reached, our test harness helps developers test for soundness violations.

We allow JAVACOP users to indicate the "stuck expressions" via a simple API for instrumenting Java bytecode with user-defined checks. In essence, the user implements a runtime checker that directly enforces the property being conservatively ensured by the pluggable type system. This runtime checker is typically quite simple relative to the pluggable type system, because it can directly inspect the dynamic program state. For example, while a practical pluggable type system for nonnull must include sophisticated reasoning based on flow sensitivity, raw types, etc., a runtime checker simply ensures that a null pointer is never dereferenced and that a `@NonNull` field or variable is only ever assigned nonnull values.

In the second stage of our testing approach, the test harness uses the given runtime checker to detect soundness violations. For each program in the test suite that successfully typechecks with the pluggable type system under test, the test harness executes that program with the user-provided instrumentation. If one of the user-defined checks fails during execution, this indicates a likely violation of type soundness in the pluggable type system.

For example, the developer of a nonnull type system could easily overlook the corner case of object construction that motivated the need for raw types in the previous section. Therefore, the developer might assume the program in Listing 12 should typecheck successfully, erroneously allowing the test case to pass the first stage of the test harness. However, in the second stage an instance of the class `B` is instantiated using the instrumented Java bytecode, causing an error to be signaled due to a null dereference and indicating the soundness violation.

6.2 Implementation

To implement a runtime checker, a JavaCOP user creates a class with one method per bytecode instruction to instrument, along with any helper methods desired. Listing 15 shows an excerpt from the runtime checker for nonnull properties. The excerpt defines two methods that respectively instrument stores into static fields and loads of instance fields. Each `test*` method takes as arguments the parameters of the bytecode that it instruments along with other useful values in context, and we use metadata annotations to identify these arguments. For example, the `testGetField` method takes as arguments the object whose field is being accessed along with the name of the field. The `test*` methods use the `generateError` method to generate a runtime error with a given message.

Given such a class, our test harness builds an adapter that instruments each bytecode instruction dynamically as each class is loaded by the JVM. We implemented this instrumentation generator on top of the ASM bytecode rewriting framework [Bruneton et al. 2002], although the developer is never required to directly manipulate bytecode. In our running example, each `putstatic` instruction is instrumented to invoke the `testPutStatic` method with the expected arguments before the instruction is executed, and similarly for the `getfield` instruction.

Instrumenting bytecode instructions is a simple solution, but it does require JavaCOP users to be familiar with bytecode and how Java source constructs translate to bytecode. This issue could be alleviated by allowing users to provide instrumentation at a higher level of abstraction, for example, providing test methods for all method invocations, all dereferences, all field updates, etc. Our framework would then automatically adapt these test methods to operate on the relevant bytecode instructions.

6.3 Experience

Building runtime checkers. We implemented complete runtime checkers for four different pluggable type systems. The full nonnull runtime checker is 36 LOC and instruments four bytecode instructions. It replicates the behavior of the Java virtual machine's null-dereference checks and additionally ensures that a `null` value is never stored in a `@NonNull` reference. Our *confined* type system [Vitek and Bokowski 1999] instrumenter consists of 37 LOC and instruments seven bytecode instructions. Confined type systems are meant to ensure that confined objects are encapsulated by their package, so the

```
public class NonnullTestMethod {

 /* check for null assigned into @NonNull field */
  public static void testPutStatic(
      @ReceiverObject Class recv,
      @FieldName String name,
      @ActionObject Object o){
    if(recv.getField(name).hasAnnotation(NonNull.class)
         && o == null)
     generateError("Cannot assign null to field "+name);
  }

  /* check for null dereference */
  public static void testGetField(
      @ReceiverObject Object recv,
      @FieldName String name) {
    if (recv == null)
      generateError("Null dereference'');
  }
}
```

Listing 15. An excerpt from the Java class that specifies the runtime semantics for a `@NonNull` type system. This excerpt signals an error when a null value is assigned into a `@NonNull` field at runtime.

instrumenter checks for dereferences and assignments of those objects outside of their packages. We also implemented runtime instrumenters for a *race-condition detection* type system [Flanagan and Freund 2000] (43 LOC, four bytecode instructions), which determines if a field is accessed without its corresponding lock being held, and for our implementation of Java's `final` class modifier (17 LOC, one bytecode instruction), which checks if a subclass of a `@Final` type is ever instantiated at runtime.

In all cases, the runtime checkers are quite simple, instrumenting only a handful of bytecode instructions and requiring only a few dozen lines of code. Furthermore, the first three runtime checkers described previously are significantly simpler than their associated pluggable type systems. Therefore, we believe that our approach provides a low-overhead mechanism for JAVACOP users to specify the intended semantics of a pluggable type system.

Using the test harness. After updating our original nonnull type system to support flow sensitivity, but before extending it to properly handle object initialization via raw types (as described in Section 5.1), we created the runtime checker for the nonnull type system. We also created a test suite consisting of 79 unit tests and used our test harness to ensure that the pluggable type system and runtime checker agree on the results of all tests.

We then used our testing framework during development of the extension to the type system to handle raw types. To do so, we first created a test case

similar to the one in Listing 12, which passed our static checker but failed the runtime checker, thereby illustrating the unsoundness. We used this test case during development to ensure that the resulting raw types checker indeed plugs the type hole. We repeated this process when adding flow sensitivity to check for definite assignment of `@NonNull` fields. Upon removing the JAVACOP rule requiring `@NonNull` fields to have initializers, the type system became unsound until the flow-sensitive checks were in place. The testing framework made it easy to concretely understand the type holes being fixed and to gauge progress toward these goals. In total we created 10 test cases when developing and testing the raw types extension to the nonnull checker.

7. JAVACOP PERFORMANCE

To demonstrate that JAVACOP is suitable for interactive development, we measured its performance compiling a range of sample programs using several pluggable type systems. The sample programs include several well-known open source examples, as well as a simple Hello World example and the Java code generated to check the rules for the JAVACOP nonnull type systems (described earlier in this article). We compiled each program first with no rules, then with a confined type system, unique reference type system, and nonnull type system individually, and then finally with JAVACOP enforcing all three of these type systems simultaneously.

The measurements were taken on a Dell Optiplex GX270, with an Intel Pentium IV 2.8GHz and 1.5GB RAM, running Fedora 8 in a KDE Konsole terminal. JAVACOP was run using the Sun Java HotSpot(TM) Client VM (build 1.6.0_05-b13, mixed mode, sharing). Each test was run five times and timed using the `time` command from the `bash` shell (i.e., "real" time was measured). The highest and lowest timings were discarded, and the remaining three averaged to produce the final figure. Because the confined type system rules rely on annotations not present in most of the examples, we modified the rules so that *every* class would be checked as if it were confined. All numbers include the time required to print warnings, errors, and `ant` output to the screen.

Figure 4 presents the results. For each configuration the upper row is time in seconds, while the lower row is the percentage slowdown over the baseline performance caused by using the given type system(s); in each case lower is better. The key point demonstrated by this table is that even when using multiple complex type systems, performing JAVACOP rule checking in addition to standard Java typechecking takes less than 1.7 times as long as Java typechecking alone. Closer inspection shows that most of the time is spent in the two type systems which use flow sensitivity: the non-null and unique type systems. This is not surprising as those rules must make multiple visits over the AST and create a number of objects representing dataflow facts. The simpler confined type system only imposes overhead of between 1% and 15 %. We have not performed any optimizations on JAVACOP , nor investigated any kind of incremental compilation support, in order to obtain these numbers. Nonetheless, JAVACOP'S performance demonstrates the practicality of its design.

System	Classes	No ruleset	Confined	Unique	Non-null	All
Hello World	1	0.6557	.7487	.7343	.7107	.7647
			14.2%	**8.4%**	**12.0%**	**16.6%**
Non-null ruleset	4	1.2930	1.4603	1.4493	1.5610	1.8147
			12.9%	**12.1%**	**20.7%**	**40.3%**
Polyglot5	161	6.1060	7.0577	8.1930	7.4580	9.3163
			15.6%	**34.2%**	**22.1%**	**52.6%**
PMD 4.2.4	786	18.1073	19.0110	20.2160	20.6807	22.9273
			5.0%	**11.6%**	**14.2%**	**26.6%**
JEdit 4.3 pre 15	1085	10.8970	11.4950	14.6067	13.5540	18.1417
			5.5%	**34.1%**	**24.5%**	**66.6%**
Jython 2.5	1191	19.2883	20.8210	24.4997	25.4143	31.6790
			7.9%	**27.0%**	**31.8%**	**64.2%**
OpenJDK 7-ea-src-b36	1281	11.0743	11.9663	14.9040	13.6270	17.4420
			8.1%	**34.6%**	**23.1%**	**57.5%**

Fig. 4. JavaCOP compilation times.

8. DISCUSSION

JavaCOP is a practical and powerful tool for building pluggable type systems. It has been used to create several interesting typecheckers not anticipated during its design. For instance, in addition to the pluggable type systems presented in this article, recent work by Fischer, Marino, Majumdar, and Millstein [2009] used JavaCOP to implement static checking for a parameterized, role-based access control system that supports fine-grained access policies. The type system includes a form of dependent types and effect checking and makes use of JavaCOP's dataflow framework to incorporate flow-sensitive reasoning. In their case study, the JavaCOP -based checker was able to check a large existing codebase (160K lines of code in 633 classes) in 11 seconds.

Through our usage of the JavaCOP suite of tools we have discovered a few important limitations. First, JavaCOP does not provide special support for parametric polymorphism over type annotations. Therefore, individual pluggable type systems must implement their own notion of type variables along with associated rules for manipulating them. JavaCOP also does not provide special support for inference of type annotations, but this can be implemented on a per-type-system basis via JavaCOP's dataflow framework.

The JavaCOP language could be improved in several ways. First, it currently supports helper predicates but not helper functions (which return values other than booleans), so code duplication is sometimes required. Second, some pluggable type systems employ annotations that contain data elements, and manipulating such parameterized annotations can be awkward and tedious. Despite this limitation, JavaCOP has been successfully used to create several type systems that employ parameterized annotations, including the type system for access control described earlier [Fischer et al. 2009], a race-condition detection type system [Andreae et al. 2006b], and a conformance checker for Enterprise Java Beans [Andreae et al. 2006b].

Finally, the JavaCOP rule language is currently quite closely tied to the OpenJDK's AST representation and is therefore fragile to small changes in

that data structure. A better approach might be for JAVACOP to use its own AST representation, allowing the JAVACOP rules to remain agnostic to the underlying implementation platform. This approach would make it easier to port JAVACOP to a different compiler framework, such as Eclipse. It would also allow us to hide some of the low-level details of the OpenJDK's AST which are currently exposed. For example, different AST nodes are used to represent initializing and non-initializing assignment statements, and JAVACOP users must be aware of this distinction in order to properly handle all possible assignments in a program.

9. RELATED WORK

An earlier version of JAVACOP was described previously [Andreae et al. 2006b]. That paper presented our declarative rule language along with a number of example pluggable type systems. This article incorporates the advances we have made in the intervening years, most notably the API for incorporating flow-sensitive reasoning and the test harness for pluggable type systems. We also present several case studies illustrating the practical utility of several pluggable type systems on existing Java code.

Papi, Ali, Correa Jr., Perkins, and Ernst [2008] have created the Checker framework for pluggable type systems in Java that is similar in many ways to JAVACOP . Both frameworks add a pass during compilation which performs the user-defined static checking; the typechecking in both systems may also employ metadata annotations. However, in the Checker framework, users directly create visitors that traverse Java's Tree API, while JAVACOP users can employ, and are encouraged to use, our declarative rule language.

The Checker framework has built-in support for inferring type annotations, which JAVACOP lacks. However, this inference mechanism only works for a special class of annotations. For example, the Checker framework can properly infer annotations for our `@NonNull` annotation but not for our `@Unique` annotation. While this inference mechanism can potentially be overridden to perform other analyses, no special support for user-defined flow analyses analogous to our dataflow framework is provided. We have found JAVACOP'S facility for easily defining a variety of flow analyses to be indispensable in building practical type systems. JAVACOP'S test harness also has no analog in the Checker framework. Finally, the Checker framework makes use of the richer annotation syntax in Java 7 [Ernst 2007], which makes pluggable type systems more expressive.

There are a wide variety of systems that provide declarative languages for imposing constraints on and querying properties of Java programs (e.g., Magellan [Eichberg et al. 2005], JQuery [Janzen and Volder 2003], SCL [Hou and Hoover 2006], CodeQuest/SemmleCode [Hajiyev et al. 2006], and JTL [Cohen et al. 2006]). Other programming languages have similar systems, such as ASTLOG [Crew 1997] and CCEL [Duby et al. 1992] for C++. These systems can be very flexible and are capable of expressing style checkers similar to our Polyglot and JUnit checkers. These systems are whole-program checkers, while JAVACOP retains the traditional modular style of Java typechecking. Finally, these systems lack an analog of our type system testing framework and provide no special support for incorporating user-defined dataflow analyses.

CQual [Foster et al. 1999] and JQual [Greenfieldboyce and Foster 2007] allow users to introduce type qualifiers in their C and Java programs, respectively. The semantics of these qualifiers is specified through subtyping relationships; users cannot provide specialized typing rules as they do in JavaCOP . This design may make it difficult to use these systems for type qualifiers whose disciplines are not based on value flow, such as our domain-specific checkers. On the other hand, both CQual and JQual support polymorphic qualifier inference, including object sensitivity for JQual, while JavaCOP requires explicit annotations.

We previously developed Clarity [Chin et al. 2005; 2006], a type qualifier system for C that allows users to provide typing rules for their qualifiers. JavaCOP brings this approach to Java and provides a much more expressive rule language: rules can constrain not just expressions but arbitrary statements, methods, and classes. Clarity 's rules were flowinsensitive, while JavaCOP incorporates a generic flow analysis framework. Clarity allows users to associate an invariant with a qualifier and uses an automatic theorem prover to validate the correctness of the qualifier's rules with respect to this invariant. The limitations of such automatic proofs led to our new testing approach to rule validation in JavaCOP .

Some of the properties that are expressible as a pluggable type system in JavaCOP can also be checked by a static analysis. For example, the FindBugs system [Hovemeyer and Pugh 2004] includes a nonnull checker, and Foster and Ma [2007] describe a static analysis for inferring uniqueness. A static analysis can typically work without user annotations. On the other hand, explicit type annotations can serve as useful program documentation, and JavaCOP rules provide an explicit discipline for programmers to think about and obey. Type annotations also enable precise modular checking, whereas static analyses often require interprocedural analysis for precision.

There are several extensible compiler frameworks for Java, for example, JastAdd [Ekman and Hedin 2004] and Polyglot [Nystrom et al. 2003]. Extensible compilers are very powerful, supporting arbitrary analyses and extensions to a language and its type system. JastAdd, for instance, provides a powerful mechanism for declarative code rewriting and analysis via extended attribute grammars. JavaCOP is tailored to the domain of pluggable type systems and so can provide specialized scaffolding, such as a declarative rule language and straightforward support for multiple type system extensions.

10. CONCLUSIONS

Pluggable type systems are a promising approach that allows developers to obtain the benefits of static typechecking for desired programming disciplines and styles. In this article we presented the design, implementation, and evaluation of the JavaCOP framework for pluggable type systems in Java. We described the JavaCOP rule language, as well as an API that supports an expressive form of flow sensitivity for pluggable type systems. We illustrated JavaCOP's benefits for enforcing both domain-specific requirements such as design patterns as well as sophisticated, flow-sensitive type systems from the research literature. Finally, we presented a novel approach to help users gain confidence in the

correctness of their JAVACOP rules and the associated type systems they are meant to implement via the JAVACOP test harness.

As future work, we are particularly interested in building on our approach to rule testing. One exciting direction is to explore automatic generation of test suites, to reduce the burden of manually creating test suites and to help ensure wider coverage. We are inspired by recent work on automatic generation of test suites for testing refactoring engines [Daniel et al. 2007] and model checking of type systems [Roberson et al. 2008].

REFERENCES

ALDRICH, J., KOSTADINOV, V., AND CHAMBERS, C. 2002. Alias annotations for program understanding. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'02)*. M. Ibrahim and S. Matsuoka, Eds. ACM Press, New York, 311–330.

ANDREAE, C., COADY, Y., GIBBA, C., NOBLE, J., VITEK, J., AND ZHAO, T. 2006a. Scoped types and aspects for real-time systems. In *Proceedings of the 20th European Conference on Object-Oriented Programming (ECOOP'06)*. Lecture Notes in Computer Science, vol. 4067. Springer,124–147.

ANDREAE, C., NOBLE, J., MARKSTRUM, S., AND MILLSTEIN, T. 2006b. A framework for implementing pluggable type systems. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'06)*. ACM Press, New York, 57–74.

ARNOLD, K., GOSLING, J., AND HOLMES, D. 2000. *The Java Programming Language*, 3rd ed. Addison-Wesley, Reading, MA.

BLOCH, J. 2002. A metadata facility for the Java programming language. Tech. rep. JSR 175. http://www.jcp.org.

BOYAPATI, C., KHURSHID, S., AND MARINOV, D. 2002. Korat: Automated testing based on Java predicates. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'02)*. P. G. Frankl, Ed. ACM Press, New York, 123–133.

BOYLAND, J. 2001. Alias burying: Unique variables without destructive reads. *Softw. Pract. Exper. 31*, 6, 533–553.

BOYLAND, J., NOBLE, J., AND RETERT, W. 2001. Capabilities for sharing: A generalisation of uniqueness and read-only. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP'01)*. J. L. Knudsen, Ed. Lecture Notes in Computer Science, vol. 2072. Springer, 2–27.

BRACHA, G. 2004. Pluggable type systems. *OOPSLA Workshop on Revival of Dynamic Languages*. http://pico.vub.ac.be/ ∼wdmeuter/RDL04/papers/Bracha.pdf.

BRUNETON, E., LENGLET, R., AND COUPAYE, T. 2002. ASM: A Java bytecode manipulation and analysis framework. In *Proceedings of the Adaptable and Extensible Component Systems Conference*. http://asm.objectweb.org.

CHALIN, P. AND JAMES, P. R. 2007. Non-Null references by default in Java: Alleviating the nullity annotation burden. In *Proceedings of the 21st European Conference on Object-Oriented Programming (ECOOP'07)*. E. Ernst, Ed. Lecture Notes in Computer Science, vol. 4609. Springer, 227–247.

CHARLES, P., GROTHOFF, C., SARASWAT, V., DONAWA, C., KIELSTRA, A., EBCIOGLU, K., VON PRAUN, C., AND SARKAR, V. 2005. X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'05)*. R. E. Johnson and R. P. Gabriel, Eds. ACM Press, New York, 519–538.

CHIN, B., MARKSTRUM, S., AND MILLSTEIN, T. 2005. Semantic type qualifiers. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05)*. V. Sarkar and M. W. Hall, Eds. ACM Press, New York, 85–95.

CHIN, B., MARKSTRUM, S., MILLSTEIN, T., AND PALSBERG, J. 2006. Inference of user-defined type qualifiers and qualifier rules. In *Proceedings of the 15th European Symposium on Programming*

Languages and Systems (ESOP'06), P. Sestoft, Ed. Lecture Notes in Computer Science, vol. 3924, Springer, 264–278.

COHEN, T., GIL, J. Y., AND MAMAN, I. 2006. JTL — The Java tools language. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'06).* ACM Press, New York, 89–108.

CREW, R. F. 1997. ASTLOG: A language for examining abstract syntax trees. In *Proceedings of the Conference on Domain-Specific Languages*. C. Ramming, Ed. USENIX Association, 229–243.

DANIEL, B., DIG, D., GARCIA, K., AND MARINOV, D. 2007. Automated testing of refactoring engines. In *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/SIGSOFT FSE'07)*. I. Crnkovic and A. Bertolino, Eds. ACM Press, New York, 185–194.

DUBY, C. K., MEYERS, S., AND REISS, S. P. 1992. CCEL: A metalanguage for C++. In *Proceedings of the USENIX C++ Conference*. USENIX Association, 99–116.

EICHBERG, M., SCHÄFER, T., AND MEZINI, M. 2005. Using annotations to check structural properties of classes. In *Proceedings of the 8th International Conference on Fundamental Approaches to Software Engineering (FASE'05)*. M. Cerioli, Ed. Lecture Notes in Computer Science, vol. 3442, Springer, 237–252.

EKMAN, T. AND HEDIN, G. 2004. Rewritable reference attributed grammars. In *Proceedings of the 18th European Conference on Object-Oriented Programming (ECOOP'04)*. M. Odersky, Ed. Lecture Notes in Computer Science, vol. 3086, Springer, 144–169.

ERNST, M. 2007. Java annotations on types. Tech. rep. JSR 308. http://www.jcp.org.

FÄHNDRICH, M. AND LEINO, K. R. M. 2003. Declaring and checking non-null types in an object-oriented language. In *Proceedings of the 18th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'03),* R. Crocker and G. L. Steele, Jr., Eds. ACM Press, New York, 302–312.

FISCHER, J., MARINO, D., MAJUMDAR, R., AND MILLSTEIN, T. 2009. Fine-Grained access control with object-sensitive roles. In *Proceedings of the 23rd European Conference on Object-Oriented Programming (ECOOP'09)*. S. Drossopoulou, Ed. Lecture Notes in Computer Science, vol. 5653. Springer, 173–194.

FLANAGAN, C. AND FREUND, S. N. 2000. Type-Based race detection for Java. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI'00)*. J. Larus and M. Lam, Eds. ACM Press, New York, 219–232.

FOSTER, J. S., FÄHNDRICH, M., AND AIKEN, A. 1999. A theory of type qualifiers. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'99).* B. G. Ryder and B. Zorn, Eds. ACM Press, New York, 192–203.

FOSTER, J. S. AND MA, K. 2007. Inferring aliasing and encapsulation properties for Java. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'07).* ACM Press, New York, 423–440.

GABRIEL, R. P., BACON, D. F., LOPES, C. V., AND STEELE JR., G. L., Eds. 2007. *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'07).* ACM Press, New York.

GAMMA, E., HELM, R., JOHNSON, R. E., AND VLISSIDES, J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Boston, MA.

GOSLING, J., JOY, B., STEELE, G., AND BRACHA, G. 2000. *The Java Language Specification*, 2nd ed. The Java Series. Addison-Wesley, Boston, MA.

GREENFIELDBOYCE, D. AND FOSTER, J. S. 2007. Type qualifier inference for Java. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'07).* 321–336.

HAJIYEV, E., VERBAERE, M., AND DE MOOR, O. 2006. CodeQuest: Scalable source code queries with data-log. In *Proceedings of the 20th European Conference on Object-Oriented Programming (ECOOP'06)*. Lecture Notes in Computer Science, vol. 4067, Springer, 2–27.

HOU, D. AND HOOVER, H. J. 2006. Using SCL to specify and check design intent in source code. *IEEE Trans. Softw. Engin. 32*, 6, 404–423.

HOVEMEYER, D. AND PUGH, W. 2004. Finding bugs is easy. *OOPSLA Companion 2004: Companion to the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems,*

Languages, and Applications. J. M. Vlissides and D. C. Schmidt, Eds. ACM Press, New York, 132–136.

IGARASHI, A., PIERCE, B. C., AND WADLER, P. 2001. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst. 23*, 3, 396–450.

JANZEN, D. AND VOLDER, K. D. 2003. Navigating and querying code without getting lost. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD'03)*. W. G. Griswold and M. Aksit, Eds. ACM Press, New York, 178–187.

JSR 302. 2006. JSR 302: Safety critical java technology. http://jcp.org/en/jsr/detail?id=302.

JUNIT. 2000. JUnit homepage. http://junit.org.

MARKSTRUM, S. 2009. Enforcing and validating user-defined programming disciplines. Ph.D. thesis, University of California, Los Angeles, CA.

MILICEVIC, A., MISAILOVIC, S., MARINOV, D., AND KHURSHID, S. 2007. Korat: A tool for generating structurally complex test inputs. In *Proceedings of the 29th International Conference on Software Engineering (ICSE'07)*. J. Knight and M. Cohen, Eds. IEEE Computer Society, 771–774.

MILLSTEIN, T. D., FROST, C., RYDER, J., AND WARTH, A. 2009. Expressive and modular predicate dispatch for java. *ACM Trans. Program. Lang. Syst. 31*, 2.

MYERS, A. C. 1999. JFlow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (PoPL'99)*. A. W. Appel and A. Aiken, Eds. ACM Press, New York, 228–241.

NYSTROM, N., CLARKSON, M. R., AND MYERS, A. C. 2003. Polyglot: An extensible compiler framework for Java. In *Proceedings of the 12th International Conference on Compiler Construction (CC'03)*. G. Hedin, Ed. Lecture Notes in Computer Science, vol. 2622. Springer, 138–152.

NYSTROM, N., QI, X., AND MYERS, A. C. 2006. J&: Software composition with nested intersection. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'06)*. G. E. Harris, Ed. ACM Press, New York, 21–36.

PAPI, M. M., ALI, M., CORREA JR., T. L., PERKINS, J. H., AND ERNST, M. D. 2008. Practical pluggable types for Java. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'08)*. B. G. Ryder and A. Zeller, Eds. ACM Press, New York, 201–212.

PIERCE, B. C. 2002. *Types and Programming Languages*. MIT Press, Cambridge, MA.

POLYGLOT. 2004. Polyglot extensible compiler framework. http://www.cs.cornell.edu/Projects/polyglot.

POLYGLOT5. 2007. Polyglot for Java 5. http://www.cs.ucla.edu/ ∼milanst/projects/polyglot5.

ROBERSON, M., HARRIES, M., DARGA, P. T., AND BOYAPATI, C. 2008. Efficient software model checking of soundness of type systems. In *Proceedings of the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'08)*. G. E. Harris, Ed. ACM Press, New York, 493–504.

TARR, P. L. AND COOK, W. R., EDS. 2006. *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'06)*. ACM Press, New York.

THOMAS, D., ED. 2006. *Proceedings of the 20th European Conference on Object-Oriented Programming (ECOOP'06)*. Lecture Notes in Computer Science, vol. 4067. Springer.

VALLÈE-RAI, R., HENDREN, L., SUNDARESAN, V., LAM, P., GAGNON, E., AND CO, P. 1999. Soot - A Java optimization framework. In *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research (CASCON'99)*. S. A. MacKay and J. H. Johnson, Eds. IBM, 125–135.

VITEK, J. AND BOKOWSKI, B. 1999. Confined types. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'99)*. B. Hailpern, L. Northrop, and A. M. Berman, Eds. ACM Press, New York, 82–96.

WARTH, A., STANOJEVIC, M., AND MILLSTEIN, T. 2006. Statically scoped object adaptation with expanders. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'06)*. P. L. Tarr and W. R. Cook, Eds. ACM Press, New York, 37–56.

WAL, A. 2007. T.J. Watson libraries for analysis (WALA). http://wala.sf.net.

Received November 2008; accepted April 2009