A Static Aspect Language for Checking Design Rules

Clint Morgan Kris De Volder Eric Wohlstadter
University of British Columbia
[clint, kdvolder, wohlstad]@cs.ubc.ca

Abstract

Design rules express constraints on the behavior and structure of a program. These rules can help ensure that a program follows a set of established practices, and avoids certain classes of errors.

Design rules often crosscut program structure and enforcing them is emerging as an important application domain for Aspect Oriented Programming. For many interesting design rules, current general purpose AOP languages lack the expressiveness to characterize them statically and enforce them at compile time.

We have developed a domain specific language called Program Description Logic (PDL). PDL allows succinct declarative definitions of programmatic structures which correspond to design rule violations. PDL is based on a fully static and expressive pointcut language. PDL pointcuts allow characterizing a wide range of design rules without sacrificing static verification.

We evaluate PDL by comparing it to FxCop, an industrial strength tool for checking design rules.

Categories and Subject Descriptors D.2.14 [*Software Engineering*]: Software/Program Verification; D.3.3 [*Programming Languages*]: Language Constructs and Features

Keywords Aspect-Oriented Programming, Design Rule

1. Introduction

Design rules constrain the structure or behavior of a program and express desirable programming practices. Examples include stylistic guidelines, library usage rules, and correctness properties.

Design rule checkers take a design rule specification, and statically check a program for violations. This allows defects to be caught early in the development process (i.e., compilation), and may catch errors that are not evident in testing. To encourage this practice, programmers must be allowed to *easily* encode design rules in a form that can be consumed by a checker.

A commonly used approach to building a design rule checker is as a framework with an API which provides a means of inspecting the program. Examples of this approach include FxCop [1] and FindBugs [18]. These rule checking frameworks come with a large number of generic, predefined rules. However, because of the complexity of the APIs and the imperative nature of the rule definitions, adding new rules is difficult.

Aspect Oriented Programming (AOP) is emerging as a more declarative alternative to imperative rule checking frameworks. Recent AOP literature has explored using AspectJ [20], a general purpose AOP language with a dynamic joinpoint model, to encode design rules but have found that it lacks the ability to express some static properties. For example, in [22], the Law of Demeter was encoded using AspectJ. Variants of this design rule are expressed in terms of static program elements and thus could in theory be checked statically. However, limitations in the pointcut and advice model in AspectJ prevented static checking. This led the authors of [22] argue for extensions to AspectJ–namely, static pointcuts and advice–which allow static aspects. Also, in [26] it is shown how aspects can be used to enforce certain design rules in a program. Several examples of how this can be done in AspectJ are given. The authors also describe some rules which cannot be expressed in AspectJ. These examples motivate the need for additional pointcuts on static program elements. The authors conclude that an AOP approach to encode and enforce designs is desirable, but AspectJ's joinpoint and pointcut mechanism does not allow for sufficient static expressibility.

In this paper we present a domain specific aspect language called Program Description Logic (PDL). PDL is designed for the specific purpose of statically checking design rules. A PDL program consists of a list of pointcut advice-pairs. Each pointcut-advice pair represents a design rule. The pointcut matches violations of a design rule, and the advice specifices a corresponding error message. Below is an example of a simple PDL design rule definition:

```
field(sourceType) && public
: "Do not declare visible instance fields"
```

The syntax of PDL pointcuts is similar to AspectJ. However, the PDL joinpoint model, unlike AspectJ's, is static. In other words, joinpoints in PDL are static program entities such as classes, fields, methods, and instructions in the byte-code. Consequently, PDL pointcuts can be fully evaluated statically. Given the purpose of PDL, the only kind of advice is emission of compile-time error messages. Thus a PDL advice body is just an error message. PDL is implemented as a bytecode tool which analyzes .NET assemblies, finds matches to the pointcuts, and prints the corresponding error messages.

To evaluate PDL, we compare it to FxCop [1], an industrial design rule checker which analyzes programs for conformance to the .NET Framework Design Guidelines [2]. FxCop is typical of the framework approach to building a design rule checker. It comes with an extensive set of prepackaged rules, but the complexity of its APIs and imperative nature make it hard to write custom rules. PDL, on the other hand, allows concise declarative rule definitions. As expected, the declarative specification limits expressibility. However, we show that we can still express a substantial and useful portion of the FxCop rules in PDL.

We also consider the performance of PDL which is an important concern for a rule checking tool. The checking process must be efficient enough that it can be placed inside of the edit-compile-

AOSD '07 March 12–16, Vancouver Canada.

debug loop that forms a programmers primary work cycle. Moreover, PDL must be able to scale up to check a large collection of rules on a large program.

Our implementation of PDL is a source to source compiler which results in minimal overhead when running the rules. Additionally, the declarative nature of PDL rules allow optimizations not otherwise possible. In particular, our implementation detects intermediate results which are shared among rules and merges the rules together to avoid repeated computations. We carried out performance measurements which shows that our implmentation of PDL performs comparably to FxCop.

The main contribution of this paper is the design and implementation of PDL, a domain specific aspect language for the specific purpose of encoding design rules. We believe this is a useful contribution by itself and also comprises a number of more specific, smaller contributions which we discuss below.

First PDL has a useful static type checker that assigns types to pointcuts based on the types of joinpoints which they can match. We believe the type checker can detect many nonsensical pointcuts without being overly restrictive. We have formalized the type-system and proven its soundness. The formalization gave rise to some interesting insights into the semantics of ! pointcuts. We believe these static typing related contributions have independent value and could be applied to other pointcut languages, for example AspectJ, which has a similar syntactic and semantic structure to untyped PDL.

Second the syntax of PDL also harbors a number of novel ideas, such as quantifiers inspired by description logics. These could be useful additions to other pointcut languages and will likely mesh well with any pointcut syntax inspired by AspectJ (i.e., the majority).

Third, we performed a detailed comparison between declarative PDL and an imperative, industrial strength design-rule checker in terms of performance and expressiveness. Other comparable work on using declarative notations for design-rules have not performed such a comparison. This is unfortunate since we believe that such industrial tools are their most important competitors in practice.

Fourth, PDL exemplifies the idea of applying database optimization techniques in the context of pointcut language optimization. While the similarity between pointcut languages and query-languages is relatively apparent and has been pointed out before (e.g., [13], [16]) real examples that exploit this idea for optimization are currently few.

The remainder of this paper is organized as follows. Section 2 provides a description of the PDL language and semantics. Section 3 discusses rule evaluation and optimization techniques. Section 4 provides a comparison between PDL and FxCop in terms of expressibility, conciseness, and performance. Section 5 presents related work. Finally, Section 6 concludes and discusses future work.

2. PDL Pointcut Language

A PDL program consists of a series of pointcut-advice pairs. Pointcuts are predicates which match joinpoints. PDL has a static joinpoint model. This means that PDL joinpoints are static program elements and that PDL pointcuts identify (match) sets of static program elements.

In this section we present the details of the PDL pointcut language, including a formal semantics and typing rules. We present the semantics in two stages. We first present a simplified, untyped version of PDL and then discuss how static typing is added.

2.1 Untyped PDL

Pointcuts in PDL are declarative expressions with a syntax inspired from AspectJ and Description Logics [4, 6]. Pointcut expressions are built up from unary and binary primitives and can be combined

$$
\begin{array}{lr}
p : Pointcut & \\
p ::= u & \text{UNARY} \\
\quad | \quad b(p) & \text{BINARY} \\
\quad | \quad p \;\&\&\; p & \text{AND} \\
\quad | \quad p \;||\; p & \text{OR} \\
\quad | \quad !\, p & \text{NOT} \\
\quad | \quad \text{exists } b\,.\,p & \text{EXISTS} \\
\quad | \quad \text{forall } b\,.\,p & \text{FORALL} \\
\quad | \quad <n\; b & \text{LESS} \\
u : Unary\ Pointcut & \\
u ::= id & \text{UNARY-PRIMITIVE} \\
\quad | \quad \text{'}pat\text{'} & \text{PATTERN} \\
b : Binary\ Relation & \\
b ::= id & \text{BINARY-PRIMITIVE} \\
\quad | \quad b+ & \text{TRANSITIVE} \\
\quad | \quad b{*}b & \text{COMPOSE} \\
id : Identifier & \\
n : Natural\ number & \\
pat : Signature\ Pattern &
\end{array}
$$

Figure 1. The syntax for a simplified version of the PDL pointcut language in BNF.

using and, or, and not operators (&&, ||, and !). There are forms for several types of quantification including existential and universal. A simplified syntax for the PDL pointcut language is shown in Figure 1.

A pointcut expression matches a joinpoint, or equivalently, defines a set of joinpoints for a given program. This section describes pointcut semantics in terms of sets of joinpoints; we also say that a pointcut *matches* a joinpoint if the joinpoint is a member of the pointcut's set. Figure 2 presents the denotational semantics of PDL pointcuts. The remainder of this section will elaborate on the syntax and semantics and provide examples.

Primitives PDL provides a number of predefined primitives. Primitives are classified as either unary or binary depending on the number of joinpoints considered for a match. Unary primitives match a joinpoint based on various properties (e.g., the access permissions of a field). Binary primitives match a joinpoint pair and express binary relations between joinpoints (e.g., the members of a type).

Unary primitives specify a set of joinpoints. The unary primitive `sourceType` denotes the set of all types declared in the program. Similarly, the `public` primitive denotes all program elements that are declared with `public` access—this includes types, fields, and methods.

A unary primitive is used in a pointcut expression by simply writing its name. The following is a simple design rule using unary pointcuts:

```
sourceType && public && nested
: "Nested types should not be visible"
```

The logical && operator connects the three primitives so that the compound pointcut will only match joinpoints that match all of the primitives (i.e., the resulting joinpoint set is the intersection of the three pointcuts).

$$\mathbf{J} := \text{The set of all joinpoints in a program}$$

$$j, j_i \in \mathbf{J}$$

$$P : Pointcut \rightarrow \mathcal{P}(\mathbf{J})$$

$$P[\![u]\!] = U[\![u]\!] \quad \text{E-UNARY}$$

$$P[\![b(p)]\!] = \{j_1 \mid (j_1, j_2) \in B[\![b]\!], j_2 \in P[\![p]\!]\} \quad \text{E-BINARY}$$

$$P[\![p_1 \;\&\&\; p_2]\!] = P[\![p_1]\!] \cap P[\![p_2]\!] \quad \text{E-AND}$$

$$P[\![p_1 \;||\; p_2]\!] = P[\![p_1]\!] \cup P[\![p_2]\!] \quad \text{E-OR}$$

$$P[\![!\; p]\!] = \{j \mid j \notin P[\![p]\!]\} \quad \text{E-AND}$$

$$P[\![\text{exists } b \;.\; p]\!] = \{j_2 \mid \exists (j_1, j_2) \in B[\![b]\!] : j_1 \in P[\![p]\!]\} \quad \text{E-EXISTS}$$

$$P[\![\text{forall } b \;.\; p]\!] = \{j_2 \mid \forall (j_1, j_2) \in B[\![b]\!] : j_1 \in P[\![p]\!]\} \quad \text{E-FORALL}$$

$$P[\![< n\; b]\!] = \{j_2 \mid \left|\{j_1 \mid (j_1, j_2) \in B[\![b]\!]\}\right| < n\} \quad \text{E-LESS}$$

$$U : Unary\ Pointcut \rightarrow \mathcal{P}(\mathbf{J})$$

$$U[\![id]\!] = \text{primitive} \quad \text{E-UNARY-PRIMITIVE}$$

$$U[\![\text{'}pat\text{'}]\!] = \{j \mid j \text{ matches } pat\} \quad \text{E-PATTERN}$$

$$B : Binary\ Relation \rightarrow \mathcal{P}(\mathbf{J} \times \mathbf{J})$$

$$B[\![id]\!] = \text{primitive} \quad \text{E-BINARY-PRIMITVE}$$

$$B[\![b+]\!] = \{(j_1, j_n) \mid (j_1, j_2), \ldots, (j_{n-1}, j_n) \in B[\![b]\!]\} \quad \text{E-TRANSITIVE}$$

$$B[\![b_1 * b_2]\!] = \{(j_1, j_3) \mid (j_1, j_2) \in B[\![b_1]\!], (j_2, j_3) \in B[\![b_2]\!]\} \quad \text{E-COMPOSE}$$

Figure 2. Denotational semantics for untyped PDL.

A binary relation expresses a relationship between two joinpoints. PDL provides a number of primitive binary relations as well as methods for creating new binary relations from existing ones (these will be presented in subsequent paragraphs).

To construct a pointcut from a binary relation b, we supply the relation a pointuct p to serve as its parameter. This is written as $b(p)$. A joinpoint j matches the binary pointcut $b(p)$ if there is a joinpoint related to j via the binary relation b that matches the parametric pointcut p.

As an example, consider the design rule *Abstract types should not have public constructors*. We encode this in PDL by writing a pointcut which matches violations—a pointcut which specifies the set of public constructors of abstract types. We use the binary relation `constructor` to relate types to their constructors. The PDL rule is shown below.

```
constructor(sourceType && abstract)
&& public
: "Abstract type should not have ..."
```

Notice that the parametric notation of binary pointcuts in PDL is similar to the `cflow` pointcut in AspectJ. The `cflow` pointcut is parametric because it takes another pointcut as a parameter. In contrast, the `call` pointcut in AspectJ takes only a method signature as a parameter. In PDL, `call` is a binary relation, so its parameter is an arbitrary PDL pointcut. This allows for the expression of more complex constraints on the callee method. For example, in the expression

```
call(method(forall field . static))
```

Name	Matches
`sourceType`	types defined in the program
`sourceNamespace`	namespaces defined in the program
`public`	public types, or members
`private`	private (internal) types, or members
`sealed`	sealed types and members
`interface`	interface types
`generic`	generic types and methods
`virtual`	virtual members
`abstract`	abstract types or members
`static`	static members
`nested`	nested types

Table 1. Selected unary primitives

the pointcut

```
method(forall field . static)
```

acts as the parameter. The semantics of the **forall** expression will be explained later. For now, we will say that the expression matches methods of types whose fields are all static. Thus, the `call` pointcut will match methods that call a method declared in a type whose fields are all static. Note that such a complex constraint is not allowed in AspectJ.

PDL defines a variety of native pointcuts that we found useful for expressing design rules. A selection of these pointcuts is described in Tables 1 and 2. Pointcuts in PDL roughly correspond to the types of operations provided in a typical API for program inspection. For example, an inspection API would have operations to examine the declaration's of a program element (e.g., check if a type is public, abstract, or sealed). These methods correspond to unary primitives in PDL. The inspection API would also provide relationships between elements (e.g., get the members of a type). These relationships correspond to binary primitives in PDL.

Many of the primitives in PDL are static versions of AspectJ pointcuts. For example, the AspectJ version of `set` matches runtime field writes, while the PDL version of `set` matches bytecodes corresponding to field writes. Similarly, the AspectJ version of `cflow` encodes a runtime relationship whereby one joinpoint is in the dynamic extent of the other. In contrast, the PDL version `cflow` uses a static approximation—reachability in the call graph.

Signature Patterns. In order for pointcuts to refer to specific types, methods, constructors, or fields, PDL provides a notation for signature pattern pointcuts. Signature patterns are written enclosed in single quotes with a form similar to that used in AspectJ. In PDL, signature patterns can be thought of as special unary pointcuts which match joinpoints based on their *signature*. For example, the expression

```
'void *.Set*(*)'
```

creates a pointcut which matches methods declared in any type, which return void, whose name starts with "Set" and takes a single argument.

Signatures are a useful mechanism to match specific elements from a set of joinpoints. For example, we often want to focus a design rule on a particular method. To encode the design rule *Finalizers should be protected*, we use a method signature to select only the finalization methods.

```
method(sourceType) && 'void *.Finalize()'
&& ! protected
: "Finalizers should be protected"
```

In AspectJ signature patterns are used as parameters to pointcuts such as `call` and `set`. However, in PDL, signature patterns

Name	Matches
`member`(p)	members declared in a type that matches p
`method`(p)	methods declared in a type that matches p
`constructor`(p)	constructors declared in a type that matches p
`field`(p)	fields declared in a type that matches p
`call`(p)	methods or bytecodes which call a method that matches p
`cflow`(p)	methods or bytecode which *could* be in the dynamic extent of a method matched by p
`within`(p)	bytecodes that implement a method which matches p
`type`(p)	types of a field, property, or namespace that matches p
`derivedType`(p)	types that are derived *from* a type which matches p
`baseType`(p)	types that are derived *by* a type which matches p
`argument`(p)	arguments of a method that matches p
`attribute`(p)	attributes of a type, method, field, property, or event that matches p
`set`(p)	bytecodes which set a field that matches p

Table 2. The semantics of selected binary primitives. Semantics are described with respect to the parametric pointcut p.

are first-class citizens—they are pointcut expressions. This is possible because PDL's joinpoint model includes the elements matched by the signature pattern (i.e., types, methods, and fields). Thus signature patterns can be used as any part of a PDL pointcut expression, as opposed to in AspectJ where they are limited to serving as parameters to a fixed number of expressions. Note that it is this generalization which allows PDL to make pointcuts like `call` and `set` parametric (as discussed previously) .

Quantification. In order to allow for more complex reasoning about joinpoints and their relationships, PDL supports several forms of pointcut quantification. The general form of these expressions was inspired from the Description Logics [4] formalism which provides a concise syntax and clean semantics for representing knowledge. In Description Logic, expressions are constructed from unary and binary predicates. This meshes nicely with PDL's concepts of unary pointcuts and binary relations.

Quantification in PDL involves checking some condition on a range of values. In practice, this allows a pointcut to match a joinpoint based on properties of related joinpoints. For example, the pointcut

```
forall field . static
```

matches types whose fields are *all* static.

Quantification expressions take a binary relation, called the range, which generates the range of the quantification. For a joinpoint j, using range R (a binary relation), the range of a quantification is defined as all values x such that $(x, j) \in R$.

An existentially or universally quantified expression takes a binary pointcut (the range), and a pointcut expression (the condition). An **exists** (similarly, **forall**) expression matches joinpoint j if one (similarly, all) of the range values x matches the condition pointcut C (i.e., $x \in C$).

Consider the design rule *Override GetHashCode() upon overriding Equals()*. A violation of the rule is a type which declares a method (i.e., there *exists* a method) for `Equals`, but does not declare a `GetHashCode` method. We can express this in PDL as

```
sourceType
&& exists method . 'bool *.Equals(object)'
&& !exists method . 'int *.GetHashCode()'
: "Override GetHashCode and Equals"
```

Another type of quantification expression is the cardinality limit. It takes a range and a natural number, and checks that the number of values in the range is below, above, or equal-to the bound. Cardinality limit expressions have the form

```
[<,>,=] n range
```

where `n` is the integer bound.

A design rule that makes use of the cardinality limit is *Namespaces should contain 5 or more types*. We can encode this with the following rule:

```
sourceNamespace && < 5 type
: "Avoid namespaces with few types"
```

The pointcut `sourceNamespace` matches all namespaces declared in the assembly. The range–`type`–is used to generate all of the types declared in a namespace, and the pointcut will match if there are less than 5 such types.

Composition. The composition of two binary relations is denoted by placing the $*$ operator between the two binary relations expressions. Thus, the expression `f*g(x)` is equivalent to `f(g(x))`. This notation can be used as syntactic sugar to remove nested parenthesis. However, a composition is required when a single binary relation is called for (e.g., as the range in an **exists** expression).

For example, to get a binary relation which matches pairs of types and the types of the fields they contain, we would write `type*field`. We use this composed pointcut to check the design rule *Types with disposable fields should also be disposable*. Checking the rule will involve looking at the type of a field (hence the composition) and is written as follows

```
sourceType
&& exists type*field
    . implements('System.IDisposable')
&& ! implements('System.IDisposable')
: "Types with disposable fields should \
 be disposable"
```

This pointcut is explained as follows. The first line will match all types declared in the program. The **exists** expression in the second and third lines will match only those types with a field which is of a type that implements the `IDisposable` interface. Finally, the last line of the pointcut matches those types which do not implement `IDisposable` and so violate the rule.

Transitive closures. The (anti-symmetric) transitive closure of a binary relation may be obtained by appending + to the expression. Transitive closures allow PDL to express a limited means of recursion. For example, the subtype relation may be obtained as the transitive closure of the `derivedType` relation:

```
derivedType+
```

This provides a convenient way to define new binary relations over the graph of certain program structures.

2.2 Typed PDL

So far we have considered all joinpoints to be of the same type. We now present a static type system for PDL that distinguishes between different types of joinpoints and assigns static types to pointcut expressions based on the types of joinpoints they can match.

Joinpoint Type	Description
namespace	Namespace
type	Type
method	Method (including constructors)
argument	Method argument
field	Field
property	Property
event	Event
attribute	Attribute of a program element
genericArgument	Type argument (to a generic type)
bytecode	Instruction in the program

Table 3. PDL joinpoint types

$$\mathbf{T} = \{\mathbf{type}, \mathbf{method}, \mathbf{field}, \ldots\}$$

$$t, t_i \in \mathbf{T}$$

$$\sigma, \sigma_1, \sigma_2 \in \mathcal{P}(\mathbf{T})$$

$$\beta, \beta_1, \beta_2 \in \mathcal{P}(\mathbf{T} \times \mathbf{T})$$

$$\frac{b : \beta \qquad p : \sigma}{b(p) : \{t_1 \mid (t_1, t_2) \in \beta, t_2 \in \sigma\}} \quad \text{T-BINARY}$$

$$\frac{p_1 : \sigma_1 \qquad p_2 : \sigma_2}{p_1 \;\&\&\; p_2 : \sigma_1 \cap \sigma_2} \quad \text{T-AND}$$

$$\frac{p_1 : \sigma_1 \qquad p_2 : \sigma_2}{p_1 \;||\; p_2 : \begin{cases} \emptyset & \text{if } \sigma_1 = \emptyset \text{ or } \sigma_2 = \emptyset \\ \sigma_1 \cup \sigma_2 & \text{otherwise} \end{cases}} \quad \text{T-OR}$$

$$\frac{p : \sigma}{!\; p : \sigma} \quad \text{T-NOT}$$

$$\frac{b : \beta \qquad p : \sigma}{\text{forall } b.p : \{t_2 \mid (t_1, t_2) \in \beta, t_1 \in \sigma\}} \quad \text{T-FORALL}$$

$$\frac{b : \beta \qquad p : \sigma}{\text{exists } b.p : \{t_2 \mid (t_1, t_2) \in \beta, t_1 \in \sigma\}} \quad \text{T-EXISTS}$$

$$\frac{b : \beta}{< n b : \{t_2 \mid (t_1, t_2) \in \beta\}} \quad \text{T-LESS}$$

$$\frac{b : \beta}{b+ : \{(t_1, t_n) \mid (t_1, t_2), \ldots, (t_{n-1}, t_n) \in \beta\}} \quad \text{T-TRANSITIVE}$$

$$\frac{b_1 : \beta_1 \qquad b_2 : \beta_2}{b_1 * b_2 : \{(t_1, t_3) \mid (t_1, t_2) \in \beta_1, (t_2, t_3) \in \beta_2\}} \quad \text{T-COMPOSE}$$

Figure 3. Typing rules for pointcuts and binary relations.

2.2.1 Typing Derivations

PDL's type system distinguishes between different types of joinpoints such as **type**, **method**, **field**, and **bytecode**. All joinpoint types supported by the current implementation of PDL are shown in Table 3. Each of these joinpoint types corresponds to a disjoint subset of joinpoints in the program. In general, a single pointcut may match joinpoints of several different types. Therefore, the type system assigns a set of joinpoint types as the type of a pointcut. Similarly, because the semantics of binary relations is in terms of pairs of joinpoints, the type of a binary relation is the pairs of joinpoint types it could match.

PDL primitives come annotated with their type. For example, the pointcut `public` has type {**type**, **method**, **field**, **property**} signifying that it matches joinpoints of type **type**, **method**, **field**, or **property**. Similarly, the binary relation `member` relates method, fields, or properties to their declaring type, and thus has type {(**method**, **type**), (**field**, **type**), (**property**, **type**)}.

In typed-PDL, a pointcut expression's type is derived by building up from the known types of the primitives. The typing rules are shown in Figure 3.

Any pointcut expression or binary relation that gets assigned $\emptyset$ as a type is considered a type error. This confirms to the intuition that pointcut expressions which could never yield results are not meaningful. A subtlety in the T-OR rule is that we have to treat $\emptyset$ as a special case to ensure that an expression which contains an erroneous subexpression is itself considered erroneous.

As an example, the type system determines the type of

```
sourceType && interface
```

by taking the intersection of the two primitive types. Thus, the pointcut has type {**type**}. While, for the pointcut

```
sourceType && virtual
```

the intersection of the two primitive types is $\emptyset$, and so PDL would generate an error message when the pointcut is compiled.

2.2.2 Soundness

It is interesting to note that the static typing rules given here are unsound with respect to the untyped semantics given in Figure 2. For example, the typing rule T-NOT is unsound. To see this, consider the pointcut `!interface`. Our type system assigns the pointcut type {**type**} (the same type as `interface`). However, the untyped semantics does not explicitly or implicitly limit the range of matched joinpoints to be of type **type**. The pointcut thus matches all kinds of joinpoints that aren't even types, for example, it matches any joinpoint which is of type **method** (since no method is an interface).

One possible way to address this issue would be to use an alternative, sound typing rule:

$$\frac{p : \sigma}{!\; p : \begin{cases} \emptyset & \text{if } \sigma = \emptyset \text{ or} \\ \mathbf{T} & \text{otherwise} \end{cases}} \quad \text{T-NOT sound}$$

We didn't adopt the above typing rule because it seems overly permissive. For example, an expression like

```
field(sourceType) && ! call('void *.foo()')
```

would pass type-checking. We felt that such expressions rather shouldn't pass type-checking since fields *never* call anything so a pointcut intent on matching fields that do not call a particular method seems ill-conceived.

Thus, instead of making the type system sound by adopting some overly relaxed typing rules we decided to keep the more restrictive typing rules and restore soundness by adopting a differ-

$$T[\![\sigma]\!] := \bigcup_{t \in \sigma} \{j \mid j : t\} \quad \text{all of the joinpoints of a given type}$$

$$P_T[\![(!\ p) : \sigma]\!] = \{j \mid j \in T[\![\sigma]\!], j \notin P_T[\![p : \sigma]\!]\} \quad \text{ET-NOT}$$

$$P_T[\![(\text{forall } b\ .\ p) : \sigma)]\!] = \{j_2 \mid j_2 \in T[\![\sigma]\!], \forall (j_1, j_2) \in B[\![b]\!] : j_1 \in P_T[\![p]\!]\} \quad \text{ET-FORALL}$$

$$P_T[\![(< n\ b) : \sigma]\!] = \{j_2 \mid j_2 \in T[\![\sigma]\!], \left|\{j_1 \mid (j_1, j_2) \in B[\![b]\!]\}\right| < n\} \quad \text{ET-LESS}$$

Figure 4. Typed semantics for PDL (only the clauses which differ from the untyped semantics are shown

ent typed semantics which is shown in Figure 4. The typed semantics restricts matched joinpoints to those that are in the static type derived for the pointcut. Not all evaluation clauses need to be changed. Intuitively, the problem arises only where the untyped semantics allow a joinpoint to match because it is *not* an element of a pointcut. In such cases we cannot assign a sound type that is more restrictive than **T** (the set of all types).

At first sight, our typed semantics may seem like "an ugly patch job". However we believe that it is actually a more intuitive semantics, in the sense that where the typed and untyped semantics differ, the typed semantics usually reflects the meaning we would expect whereas the untyped semantics does not. For example, consider the pointcut

```
abstract && ! implements('ISerializable')
```

Under the typed semantics it matches abstract types that do not implement an `ISerializable` interface. However, under the untyped semantics it also matches all abstract methods.

With the typed semantics we can formulate the following soundness theorem.

THEOREM 1 (Soundness).

$$p{:}\sigma, \sigma \neq \emptyset \quad \Rightarrow \quad P_T[\![p{:}\sigma]\!] \subset T[\![\sigma]\!]$$

The proof is straightforward by structural induction on typing derivations.

3. Implementation

This section provides an overview of the implementation of the PDL rule checker. The checker takes a set of PDL design rules and looks for violations (matches to the pointcuts) in a .NET assembly. Our implementation is a source-to-source compiler. The compiler performs type checking and optimization. The output of the compiler is C# code which performs the analysis encoded in the PDL rules. The resulting C# code is then compiled to bytecode and the analysis is run on a program to find violations of the design rules.

We see the following advantages to our implementation. First, by generating (relatively straightforward) imperative code we achieve good performance when the code is compiled to bytecode. The code we generate also provides a convenient way to inspect the output of the pointcut-compiler to verify its correctness. Second, our primitives are defined in terms of imperative methods which makes it relatively easy to extend the language with new primitives. Finally, because we often have multiple options on how to generate code, the compiler has flexibility to chose an efficient plan to perform the analysis.

3.1 Code Generation

The C# code produced by the compiler makes calls to an abstract interface for program introspection. The introspection interface represents various types of program elements (e.g., types, methods, fields) and provides methods for examining their properties and relationships. The introspection interface was defined to allow multiple back-ends to implement it. We currently have a complete implementation using Microsoft CCI[1], and partial implementations using Microsoft Phoenix[2], and Mono Cecil[3]. A back-end is specified by the user at execution time.

To implement PDL pointcuts with imperative code, we represent joinpoints as objects in the introspection API, and use methods of these objects to implement the primitive pointcuts. PDL allows primitives to be implemented by a number of different strategies. For example, the unary primitive `sourceType` can be implemented with a method that generates all of the types declared in the program, or with a method that takes a type as a parameter and returns a boolean representing if it was declared in the program. This allows the compiler to chose an appropriate method to call depending on the context. For example, in a context where types are being enumerated for another reason, code that does something with source types could be

```
foreach (Type t in ...)
   if (IsSourceType(t))
      // Do something with source type
```

Whereas, in another context source types could be enumerated from scratch

```
foreach (Type t in SourceTypes())
   // Do something with source type
```

The PDL implementation makes it easy to define new primitives. PDL has an extension API with which to register a binding between a name of a primitive and a method in the introspection API that provides an implementation of the primitive. No additional meta-data needs to be provided. All necessary information (e.g., type information for the purpose of type checking) is derived from the signature of the method.

To compile a pointcut, we must first decide on a plan for analyzing the program to find the matching joinpoints. We call this intermediate representation an *execution plan*. Its structure and interpretation is similar to the operator tree representation used in databases [7]. In PDL, an execution plan is a graph-based model of the analysis required to find matches to a joinpoint. Edges represent streams of joinpoints, and nodes transform the streams.

A given pointcut may have many equivalent execution plans, and optimization techniques may be used to determine the most efficient plan. The main optimization done by the current implementation of PDL is to merge shared subexpression. This is described in the next section.

To compile an execution plan, we generate C# code which performs the encoded analysis. For example, the PDL pointcut

```
method(sourceType && sealed) && virtual
```

is compiled to code similar to the following (simplified for presentation):

```
foreach (Type t in SourceTypes())
  if (t.IsSealed())
    foreach(Method m in t.Methods())
```

[1] Included as part of the FxCop framework

[2] `http://research.microsoft.com/phoenix/` (verified September 2006)

[3] `http://www.mono-project.com/Cecil` (verified September 2006)

```
if (m.IsVirtual())
    // Record m as violation
```

3.2 Optimization

There is a similarity between PDL rule checking and database query evaluation–PDL pointcuts are program queries, and rule checking amounts to accessing the program to find elements which match the query. Because PDL rules are often run at the same time in a large batch, we can use techniques from database multiple-query optimization [25] to improve performance.

Multiple-query optimization involves examining the execution plan to detect subtrees that produce the same result. Then, the two pieces are merged together to avoid a redundant calculation.

As an example, consider the design rules *Do not declare virtual methods on sealed types*, and *Do not declare protected methods on sealed types*. We encode these rules as

```
method(sourceType && sealed) && virtual
: "rule1"
method(sourceType && sealed) && protected
: "rule2"
```

The merging optimization detects the redundancy in computing methods of sealed types. This computation is then merged to generate code like the following:

```
foreach (Type t in SourceTypes())
  if (t.IsSealed())
    foreach(Method m in t.Methods()) {
      if (m.IsVirtual())
          // Record m as a violation to rule1
      if (m.IsProtected())
          // Record m as a violation to rule2
    }
```

Rule checking frameworks typically achieve a similar effect of sharing by implementing rules as visitors to certain code elements. The framework will iterate through each code element, and pass the element to the `Visit()` method for each rule. Thus, the cost of retrieving each element is amortized over all of the rules.

In a typical framework, the visit-able elements are fixed to a small number of basic code elements (e.g., types, methods and fields). However, there is often sharing between rules at a granularity not supported by the framework. For example, imagine there are several rules which govern the way interface `I` should be implemented. In such a case, there are two options to encode the rules using the visitor pattern:

1. Implement each rule as a separate type visitor. Each visitor first checks if the type implements `I`, the does the rest of its checking.
2. Combine all of the rules into a single type visitor. When checking a type, the visitor first checks whether the type implements `I`. If so, the combined rule can check each of its constituent rules.

The first option maintains modularity (the rules are all separately encoded), while sacrificing performance (each rule separately calculates the same condition). The second option makes the opposite trade-off.

By using a declarative language like PDL, combined with the merging optimization, we can achieve *both* modularity and performance. We can encode the rules separately and rely on the compiler to weave them together.

Category	# rules	# in PDL	ratio
Design	59	39	.66
Globalization	7	2	.29
Interoperability	16	2	.13
Mobility	2	0	0
Naming	26	2	.08
Performance	22	4	.18
Portability	3	0	0
Security	26	5	.19
Usage	40	20	.50
Total	201	74	.37

Table 4. Overview of the FxCop rules encoded in PDL.

4. Comparison with FxCop

We evaluate PDL by comparing it to FxCop[4], an industrial strength checker for the .NET Framework. FxCop comes with 201 predefined rules most dealing with the .NET Framework Design Guidelines [2]. Rules in FxCop are implemented in C# using a visitor pattern.

We use FxCop to provide several points of comparison with PDL. First, we investigate expressibility. Using a declarative language like PDL, we expect a loss of expressibility over a general-purpose approach. Comparing with FxCop allows us to quantify this loss, in terms of the fraction of FxCop rules that we can express in PDL. Next, we investigate the conciseness; we expect the declarative approach of PDL to result in more compact rule definitions. Then, we compare precision, how well the PDL version of a rule corresponds to the FxCop version. Finally, we compare the performance of the two tools.

4.1 Expressiveness

To evaluate the expressiveness of PDL, we went through all of the FxCop rules and attempted to encode them in PDL. Some rules were straightforward to encode in PDL, while others were obviously not suitable.

The results of this process are shown in Table 4. In total there are 201 rules defined in FxCop divided into 9 sections. We encoded 74 of the FxCop rules in PDL. PDL is best able to capture the design and usage rule categories; in both of these categories we are able to encode at least half of the FxCop rules.

While we are able to express 74 of the FxCop rules, we must emphasize that the analysis encoded in PDL sometimes differs from that encoded by FxCop. This is not unexpected because design rule checkers like FxCop and those defined with PDL are essentially opportunistic in nature. Rather than actually encoding the truly interesting property, which is often dynamic and intractably hard to verify statically, design rule checkers are opportunistic in that they encode an analysis which checks some approximation of the interesting property that is easily checkable with the available machinery. In some cases, the PDL version is closer to the "spirit" of the design rule, and in other cases, FxCop does a better job. For an example of a rule where FxCop performs a more accurate analysis consider the rule *Dispose methods should call base class dispose*. The analysis encoded in PDL simply check that there exists a call to the base class method within the dispose method. However, FxCop uses a more precise data and control flow analysis to determine if the base class method is called on *all* paths through the method. Examples where PDL is more accurate are those using a `cflow` pointcut. FxCop uses a fairly crude approximation by

[4] Version 1.35

Category	Total	J	P	C	O
Design	20	6	7	7	0
Globalization	5	1	0	4	0
Interoperability	14	0	13	1	0
Mobility	2	0	0	2	0
Naming	24	2	4	14	4
Performance	18	3	8	4	3
Portability	3	0	1	0	2
Security	21	2	13	6	0
Usage	20	3	9	8	0
Total	127	17	55	46	9

Table 5. Overview of the reasons PDL was unable to express FxCop rules. Column **J** is due to the joinpoint model, **P** is due to a lack of primitives, **C** is due to a complex analysis, and **O** is other.

considering call-chains of at most length 2 whereas PDL considers call-chains of any length.

Of the rules that PDL cannot express, there are three general reasons—either the rule deals with elements missing from PDL joinpoint model, PDL lacks the primitives to express an analysis, or there is an intrinsic complexity in the rule that was not expressible in PDL. Table 5 provides a breakdown of the FxCop rules that PDL could not express.

Missing joinpoint type. Some FxCop rules deal with program elements for which there is no corresponding joinpoint in PDL. This includes rules dealing with assemblies, local variables, and literals. Naturally, PDL is unable to express these rules.

Lack of primitives. Some of the FxCop rules deal with program elements which PDL has joinpoints for, however PDL currently lacks the primitives to check the relevant properties of the joinpoints. An example of this sort of rule is *Do not catch general exception types* which requires checking the type of a catch expression to determine if it is of type `Exception` or `ApplicationException`. PDL lacks primitives which can pick out catch exceptions and examine their type, and so it is impossible to express this rule.

This could be remedied by adding the appropriate primitives. One option would be to add a binary primitive `catches` where `catches`(p) matches a bytecode catch instruction if the type it catches matches pointcut p. Then the rule could be expressed as

```
within(sourceType)
&& catches('Exception'
          || 'ApplicationException')
: "Do not catch general exceptions"
```

Excessive complexity. Some of the FxCop rules fall into the broad category of excessive complexity. Examples of such rules range from spell checking and parsing, to more complex relationships between multiple program elements. The majority of FxCop's naming rules require spell and case checking on identifiers, and so were not expressible in PDL.

As an example of a complex relationship not expressible in PDL, consider the rule *Members should differ by more than return types*. Checking this rule involves checking pairs of methods to determine if they have the same argument types in the same order. Because PDL reasons about joinpoints in terms of sets, there is no way to capture this ordering relationship.

4.2 Conciseness

We evaluate conciseness by comparing the length of the rules encoded in PDL and FxCop. Of the 74 rules in PDL, the average length is 4 lines (including the line for the error message). Because we do not have the source of FxCop available, we cannot measure the length of the FxCop encodings. However, we can, with the help of a disassembler, examine the rule assembly and get an approximate idea of the length. We conclude that the smallest FxCop rules are on the order of 10 lines, and the largest on the order of several hundred. Thus, as expected, it is clear that the declarative PDL encodings are much more concise than those of imperative FxCop.

4.3 Precision

As was mentioned previously, there are often semantic differences between the encoding of the same rule in FxCop and PDL. To investigate the consequences of these differences, we checked the 74 rules on the Spring.Net Framework [3] using each tool and compared the results. FxCop reported 110 violations, while PDL reported 147. The two tools agreed on 109 violations, PDL reported 38 that FxCop did not, and FxCop reported 1 that PDL did not.

Of the 38 violations that PDL alone reports, 17 were due to the fact that FxCop was able to determine that the program uses a runtime version without support for generics, and thus turn off rules which suggest generic versions of types and methods. These "false positives" could be filtered out in PDL if such information about the assembly was exposed.

The remaining 21 violations were genuine false positives. FxCop was able to use additional logic to avoid reporting the violations, while PDL was not able to express the additional logic due to the limitations on expressiveness presented in Section 4.1.

The violation reported by FxCop and not by PDL was a valid violation that PDL was not able to detect because it dealt with a special case of a rule that examined the string value of an argument.

4.4 Performance

We compare performance of PDL and FxCop. The Microsoft CCI back-end is used for PDL, which is the same back-end used by FxCop. We measure the time it takes to evaluate a set of rules ignoring the time to initialize the checking infrastructure and load the assembly. This measurement does not reflect the time needed to compile the PDL rules which could be done once and amortized over subsequent checks.

Our first set of experiments measures the time to run the 74 rules on programs of various sizes. The results are shown in Table 6. Compared to FxCop, PDL apparently executes considerably faster. However it should be acknowledged that it is hard to attribute this speed-up completely to the superiority of PDL's implementation. As pointed out earlier there are semantic differences for some rules expressed in PDL versus FxCop. These semantic differences likely account for some of the performance difference. Also, FxCop does some additional work, such as storing data to later generate a report. We believe these functionalities only marginally contribute to the difference in running time, but because FxCop is closed-source we had no way of disabling or otherwise objectively measuring the effects of these features on FxCop's running time.

We also note that the merging optimization increases the performance of PDL by around 10 percent.

In order to measure scalability of PDL with respect to the rule set size and the effect of the merging optimization, we vary the number of rules being evaluated from 1 to 74. For each size n, we randomly select 50 samples of size n from the complete rule set. We then check these rules on mscorlib using PDL with and without optimizations. The times reported for each size is the average of the 50 samples.

The results are shown in Figure 5. As expected, the merging optimization becomes more effective as more rules are added to the system. This is because, as the number of rules increases, there

Assembly	Size	PDL	PDL-opt	FxCop
Spring.Core	296K	2.4	2.1	10.7
SharpDevelop.Core	1.3M	7.7	7.0	32.7
.NET mscorlib	4.2M	9.4	8.4	72.6

Table 6. Performance comparison of PDL and FxCop on several different programs. The numbers shown are time in seconds (averages of 10 runs).

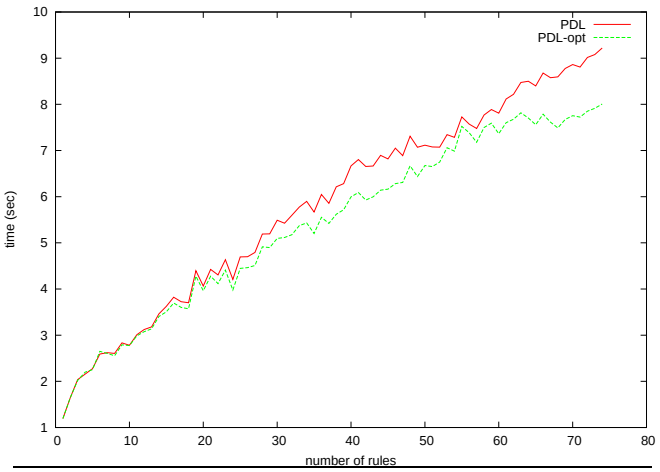

Figure 5. Performance comparison between PDL with and without optimizations as a function of the number of rules.

is a greater number of intermediate results, and hence a greater opportunity for merging.

There is an performance anomaly between sizes 45 and 65 where the speedup in the optimized version is almost lost. We attribute this effect to differing memory usage patterns[5] which staggers the effect of the .NET garbage collector's adaptations (which we are unable to control).

5. Related Work

There is a large body of work that deals with tools to check and verify programs to find errors. Approaches can be distinguished based on whether they perform static or dynamic checking. Example of an approach that uses dynamic checking are `jmlc` and`jmlunit` [21]. We are mostly interested in static techniques. In the discussion below we distinguish static approaches based on whether their emphasis is on (static verification of) dynamic properties or structural properties. The former tend to allow developers to capture interesting behavioral properties more precisely but scale less well to larger code-bases. Our work falls in the latter category.

In the first category we find approaches which use model checking and automated theorem proving techniques. For example SLAM [5], ESC/ESCJava [11, 15], Vault [10], Alloy [19], and PQL [23]. Compared to our work, these approaches are more ambitious with respect to the kinds of properties they intend to verify and how these properties are specified by the developer. Each approach provides different notations for specifying constraints that should hold over the execution of a program and provides an analyzer that attempts to statically verify whether these execution-time constraints hold. While such an approach allows developers to more directly and precisely express dynamic properties they are interested in, the static analyzers tend to be complex and scale less well to larger code-bases. Our approach is radically opposed to these approaches. PDL has a purely static joinpoint model and takes the premise that PDL rules are assertions about static code elements rather than runtime behavior. Thus, a developer using our language cannot directly express runtime properties. Instead they may be able to code-up a PDL rule that is a static approximation of it that is guaranteed to be relatively easy to check. In this way PDL trades off expressive power and precision for scalability.

Examples of approaches which are similar to PDL and share its emphasis on static/structural properties are SABER [24], FindBugs [18], FxCop [1], Metal [17], ASTLog [9], SOUL [27], and JTL [8].

SABER [24], FindBugs [18], and FxCop [1] are rule-checking frameworks which allow developers to implement checkers in an imperative language. The study presented in this paper compares PDL to FxCop. It shows how PDL trades off flexibility to express a wider variety of structural properties for concise definitions and more efficient checking. We believe FxCop is representative for this type of approach.

Other approaches that, like PDL, have taken a declarative approach to reasoning about a program's structure are ASTLog [9], SOUL [27] and JTL [8]. SOUL and ASTLog use a declarative language that is similar in syntax and expressiveness to Prolog (i.e., Turing complete). Similarly, GENOA [12] uses a domain specific language to generate customized program analyses for C++ code. A subset of GENOA has be proven to generate analyses which execute in polynomial time.

JTL [8] is a logic language for querying Java programs. It is probably the approach that is most similar in nature to PDL. There is strong similarity between the underlying semantics of both JTL and PDL. However the syntax of JTL and PDL are quite different and follow a different philosophy. JTL's syntax is designed to mimic the code elements which they match. PDL syntax is designed to be similar to AspectJ syntax. We believe that because of the differences, some rules would be easier to express with PDL whereas other would be easier to express in JTL but we have not performed a detailed comparison to try to quantify the practical impact of this.

In Metal [17], rules are expressed as state machines which are applied to the execution paths of each function in a program. This allows for a natural definition of, for example, rules of the form call X after calling Y which makes behavioral rules more natural. This is contrasted with PDL where rules are essentially queries over structural properties of the program.

As we noted in Section 1, researchers are beginning to explore AOP as a mechanism to express and enforce design rules. In addition to the work on AspectJ discussed earlier, [14] presents a checker built as a framework on top of a library for AOP. In this system, rules are encoded imperatively as type visitors which make calls to the AOP interface to inspect the class. In contrast, PDL provides a declarative approach.

Other research has focused on the performance of program queries. In [16], a DataLog language for program queries and an accompanying database system is presented. The system, CodeQuest, aims to efficiently execute DataLog queries over a program by storing the relevant relations in a database and leveraging existing database technology. We see this work as complimentary to PDL, as it would be possible to target CodeQuest as another PDL back-end. Currently, the back-ends for PDL are all introspection engines.

6. Conclusion and Future Work

We have presented PDL, an aspect language to declaratively encode design rules. PDL was inspired from the pointcut language in AspectJ, with syntactic additions reminiscent of description logics.

[5] The optimized version creates fewer objects, with longer lifetimes.

PDL introduces a type system to detect meaningless pointcuts and modify the semantics to provide a more appropriate behavior.

We demonstrated that PDL can concisely express a non-trivial subset of the rules checked by FxCop. Performance experiments show that our approach is comparable to FxCop, and can scale to large codebases.

There are several opportunities for future work on PDL. The results of our evaluation suggest that the expressibility of PDL would benefit by adding new types to the joinpoint model, and expanding the set of primitives. Additionally, we plan to explore using PDL to encode application-specific rules. We believe that using a concise, declarative language like PDL could allow the developer to encode design rules as they are encountered in the code. Finally, there is much room for optimizations in the PDL compiler. For example, we plan on expanding our search through the space of equivalent plans to discover plans which are more efficient execution or allow additional opportunities for merging.

Acknowledgments

This work was supported by a Microsoft Phoenix – Excellence in Programming Award and by the University of British Columbia.

References

[1] FxCop Team Page. `http://gotdotnet.com/team/fxcop`, verified September 2006.

[2] .NET Framework Design Guidelines. `http://msdn2.microsoft.com/en-us/library/ms229042.aspx`, verified September 2006.

[3] Spring .NET Application Framework. `http://www.springframework.net`, verified September 2006.

[4] Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2003.

[5] Thomas Ball and Sriram K. Rajamani. The SLAM project: debugging system software via static analysis. *Proceedings of the 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 02)*, 37(1):1–3, January 2002.

[6] Alexander Borgida. Description logics in data management. *IEEE Transactions on Knowledge and Data Engineering*, 7(5):671–682, 1995.

[7] Surajit Chaudhuri. An overview of query optimization in relational systems. In *PODS '98: Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 34–43, New York, NY, USA, 1998. ACM Press.

[8] Tal Cohen, Joseph Gil, and Itay Maman. Jtl: the java tools language. In Peri L. Tarr and William R. Cook, editors, *OOPSLA*, pages 89–108. ACM, 2006.

[9] Roger F. Crew. ASTLOG: A language for examining abstract syntax trees. In *In Proceedings of the USENIX Conference on Domain-Specific Languages*, pages 229–242, 1997.

[10] Robert DeLine and Manuel Fahndrich. Enforcing high-level protocols in low-level software. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 59–69, 2001.

[11] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Technical Report Technical Report TR SRC-159, COMPAQ SRC, Palo Alto, USA, 1998.

[12] Premkumar T. Devanbu. GENOA — a customizable, front-end-retargetable source code analysis framework. *ACM Transactions on Software Engineering and Methodology*, 8(2):177–212, 1999.

[13] Michael Eichberg, Mira Mezini, and Klaus Ostermann. Pointcuts as functional queries. In Wei-Ngan Chin, editor, *Programming Languages and Systems: Second Asian Symposium, APLAS 2004*, Lecture Notes in Computer Science, pages 366–382, Taipei, Taiwan, November 2004. Springer-Verlag Heidelberg.

[14] Michael Eichberg, Mira Mezini, Thorsten Schafer, Claus Beringer, and Karl Matthias Hamel. Enforcing system-wide properties. In *ASWEC '04: Proceedings of the 2004 Australian Software Engineering Conference (ASWEC'04)*, page 158, Washington, DC, USA, 2004. IEEE Computer Society.

[15] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI'2002)*, volume 37:5, pages 234–245, June 2002.

[16] Elnar Hajiyev, Mathieu Verbaere, Oege de Moor, and Kris De Volder. Codequest: querying source code with datalog. In *OOPSLA '05: Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 102–103, New York, NY, USA, 2005. ACM Press.

[17] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific static analyses. In *ACM Conference on Programming Language Design and Implementation*, pages 69–82, June 2002.

[18] David Hovemeyer and William Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12):92–106, 2004.

[19] Daniel Jackson. Alloy: a lightweight object modelling notation. *Software Engineering and Methodology*, 11(2):256–290, 2002.

[20] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.

[21] G. Leavens and Y. Cheon. Design by contract with JML, 2003.

[22] Karl Lieberherr, David H. Lorenz, and Pengcheng Wu. A case for statically executable advice: checking the law of demeter with aspectj. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 40–49, New York, NY, USA, 2003. ACM Press.

[23] Michael Martin, Benjamin Livshits, and Monica S. Lam. Finding application errors and security flaws using pql: a program query language. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 365–383, New York, NY, USA, 2005. ACM Press.

[24] Darrell Reimer, Edith Schonberg, Kavitha Srinivas, Harini Srinivasan, Bowen Alpern, Robert D. Johnson, Aaron Kershenbaum, and Larry Koved. Saber: smart analysis based error reduction. In *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, pages 243–251, New York, NY, USA, 2004. ACM Press.

[25] Timos K. Sellis. Multiple-query optimization. *ACM Trans. Database Syst.*, 13(1):23–52, 1988.

[26] Mati Shomrat and Amiram Yehudai. Obvious or not?: regulating architectural decisions using aspect-oriented programming. In *AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development*, pages 3–9, New York, NY, USA, 2002. ACM Press.

[27] Roel Wuyts. Declarative reasoning about the structure object-oriented systems. In *Proceedings of the TOOLS USA '98 Conference*, pages 112–124. IEEE Computer Society Press, 1998.