JTL – the Java Tools Language

Tal Cohen Joseph (Yossi) Gil * Itay Maman

Department of Computer Science
Technion—Israel Institute of Technology
Technion City, Haifa 32000, Israel
ctal, yogi, imaman @ cs.technion.ac.il

Abstract

We present an overview of JTL *(the Java Tools Language, pronounced "Gee-tel"), a novel language for querying* JAVA *[8] programs.* JTL *was designed to serve the development of source code software tools for* JAVA*, and as a small language to aid programming language extensions to* JAVA*. Applications include definition of pointcuts for aspect-oriented programming, fixing type constraints for generic programming, specification of encapsulation policies, definition of micro-patterns, etc. We argue that the* JTL *expression of each of these is systematic, concise, intuitive and general.*

JTL *relies on a simply-typed relational database for program representation, rather than an abstract syntax tree. The underlying semantics of the language is restricted to queries formulated in First Order Predicate Logic augmented with transitive closure (*FOPL**).*

Special effort was taken to ensure terse, yet readable expression of logical conditions. The JTL *pattern* `public abstract class`*, for example, matches all abstract classes which are publicly accessible, while* `class { public clone(); }` *matches all classes in which method* `clone` *is public. To this end,* JTL *relies on a* DATALOG*-like syntax and semantics, enriched with quantifiers and pattern matching which all but entirely eliminate the need for recursive calls.*

JTL*'s query analyzer gives special attention to the fragility of the "closed world assumption" in examining* JAVA *software, and determines whether a query relies on such an assumption.*

The performance of the JTL *interpreter is comparable to that of* JQuery *after it generated its database cache, and at least an order of magnitude faster when the cache has to be rebuilt.*

Categories and Subject Descriptors D.3.0 [*Programming Languages*]: General; D.2.3 [*Software Engineering*]: Coding Tools and Techniques

General Terms Design, Languages

Keywords Declarative Programming, Reverse Engineering

* Research supported in part by the IBM faculty award

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

OOPSLA'06 October 22–26, 2006, Portland, Oregon, USA.
Copyright © 2006 ACM 1-59593-348-4/06/0010. . . $5.00.

1. Introduction

The designers of frontier programming paradigms and constructs often choose, especially when static typing is an issue, to test-bed, experiment, and even fully implement the new idea by an extension to the JAVA programming language. Examples include ASPECTJ [52], JAM [5], Chai [71], OpenJava [76], the host of type systems supported by pluggable type systems [6], and many more.

A prime component in the interaction of such an extension with the language core is the mechanism for selecting program elements to which the extension applies: The pointcuts of ASPECTJ, for example, select the point in the code on which an aspect is to be applied. Also, a key issue of implementing the conclusions of a genericity treatise [31,51] is checking whether a given set of classes are legible as parameters for a given generic construct—what the community calls a *concept*.

JTL (the *Java Tools Language*) is a declarative language, belonging in the logic-programming paradigm, designed for the task of selecting JAVA program elements. Two primary applications were in mind at the time when the language was first conceived:

(a) Join-point selection for aspect-oriented programming, where JTL can serve as a powerful substitute of ASPECTJ's pointcut syntax, and,

(b) expressing the conditions making up concepts, and in particular multi-type concepts, for use in generic programming.

As JTL took shape and grew older it became clear it can be used not only for language extension, but also for other software engineering tasks, primarily as a tool to assist programmers understand the code they must modify. This particular problem of program understanding even if it is far from being entirely solved by JTL, is dear to our hearts: First, software development activities in industry include (and probably more and more so) the integration of new functionalities in existing code Second, maintenance remains a major development cost factor.

JTL's focus is on the modules in which the code is organized, packages, classes, methods, variables, including their names, types, parameters, accessibility level and other attributes. JTL can also inspect the interrelations of these modules, including questions such as which classes exist in a given unit, which methods does a given method invoke, etc. Additionally, JTL can inspect the imperative parts of the code by means of dataflow analysis. The extension of JTL to deal with with control-flow aspects of the code is left for further research.

1.1 Three Introductory Examples

JTL syntax is terse and intuitive; just as in AWK [1], one-line programs are abundant, and are readily wrapped within a single string. In many cases, the JTL *pattern* for matching a JAVA program

element looks exactly like the program element itself. For example, the JTL *predicate*[1]

```
public abstract void ()
```

matches all methods (of a given class) which are abstract, publicly accessible, return **void** and take no parameters. Thus, in a sense, JTL mimics the *Query By Example* [80] idea.

As in the logic paradigm, a JTL program is a set of predicate definitions, one of which is marked as the program goal.

Even patterns which transcend the plain JAVA syntax should be understandable, e.g.,

```
abstract class {
   [long | int] field;
   no abstract method;
}
```

matches abstract classes in which there is a field whose type is either **long** or **int** and no abstract methods.

The first line in the curly brackets is an existential quantifier ranging over all class members. The second line in the brackets is a negation of an existential quantifier, i.e., a universal quantifier in disguise, applied to this range.

JTL can also delve into method bodies, by means of intra-procedural dataflow analysis, similar to that of the class file verifier. Consider for example *cascading methods*, i.e., methods which can be used in a cascade of message sends to the same receiver, as in the following JAVA statement

```
(new MyClass()).f().g().h();
```

in which `f`, `g` and `h` are methods of `MyClass`. Then, the following JTL pattern matches all cascading methods:

```
instance method {
   all [ !returned | this ];
}
```

The curly brackets in the above pattern denote the set of values (including temporaries, parameters, constants, etc.) that the method may generate or use. The statement inside the brackets is a requirement that all such values are either not returned by the method, or are provably equal to **this***, and therefore guarantees that the only possible value that the method may return is* **this**.

1.2 Applications

The JTL interpreter (work on the compiler is in progress) can be used in two ways: (i) as a stand-alone program; (ii) as a JAVA library, an API, to be called directly from within any JAVA program, in a manner similar to SQL . This API makes JTL useful in implementing not only language extensions (such as a pointcut evaluation engine for an AOP compiler), but also in various kinds of software engineering tools, including searches in programs [61], LINT-like tools [29, 48, 68], and pattern identification [33].

JTL's ability to generate output text based on matched program elements enables the use of JTL not only for *search* operations in programs, but also for *search-and-replace* operations. A key applications for this ability is the context-aware renaming of program elements [30].

The brevity of expression in JTL makes it suitable for integration as an interactive search language in an IDE. Also, a JTL configuration file can specify to a compiler program elements on which special optimization effort should be made, exclusion from warnings, etc. Thus, what makes JTL unique as a "tool for making tools", are its suitability for direct user interaction, and begin a small language for configuring other tools.

To an extent, JTL resembles TCL [65], the general purpose Tool Command Language, except that JTL concentrates on language processing.

In evaluating JTL suitability for other applications we wrote several collections of JTL patterns, the two largest being (i) the implementation of the entire set of μ-patterns developed by two of us [33] and (ii) the implementation of the entire set of Eclipse[2] and PMD[3] warning messages (with the exclusion of warnings pertaining to source formatting and comments, since the current implementation of JTL works on compiled classes). Other applications for which we used JTL included aspect pointcuts, and preconditions for mixins and generics.

An Eclipse plugin for running JTL queries over JAVA projects was implemented, and its performance was compared with that of a similar plugin which uses JQuery [50] for querying JAVA code.

1.3 Underlying Model

Underlying JTL is a *conceptual* representation of a program in a simply-typed relational database. The JTL user can think of the interrogated JAVA program as a bunch of program elements stored in such a database.

JTL is declarative, sporting the simple and terse syntax and semantics of logic programming for making database queries. JTL augments these with a number of enhancements that make it even more concise and readable. Predicates are the basic programming unit. The language features a set of native predicates (whose implementation is external to the language), with a library of pre-defined predicates built on top of the native ones. Many of the native and pre-defined predicates are conveniently named after JAVA keywords.

Thus, the scheme of this database is defined by the set of native JTL predicates. Standard library predicates and user-defined predicates, defined on top of the natives, can be thought of as database *views* or *pre-defined queries*.

Interestingly, JAVA (and many other software systems) are best modeled as *infinite* databases. The reason is that in JAVA and in almost all programming languages, one cannot hope to obtain all user code which uses a certain code-bit stored in a software library. Similarly, the list of classes that inherit from a given class is unbounded. This is quite the opposite of traditional database systems, which rely on a finite, closed-world, model.

The JTL processor analyzes the queries presented to it, determining whether they are open-ended, i.e., the size of the result they return is unbounded. (In practice, only a finite approximation of the infinite database is stored. An open-ended query can be thought as a query whose size increases indefinitely with that of the approximation.)

Note that this conceptual representation does not dictate any concrete representation for the JTL implementation. JTL is applicable to several formats of program representation, ranging from program source code, going through AST representations, JAVA reflection objects, BCEL[4] library entities, to strings representing the names of program elements. In fact, JTL's JAVA API is characterized by input- and output- data representation flexibility, in that JTL calls can accept and return data in a number of supported formats.

We stress that JTL can be implemented in principle on top of any source code parser, including the JAVA compiler itself.

Two central concerns in the language design were *scalability* and the *simplification* of the idiosyncracies of logic programming.

We found, in accordance with the experience reported by Hajiyev, Verbaere and More with their *CodeQuest* system [44], that the underlying relational model, together with combinations of bottom-up and top-down evaluation strategies that DATALOG [17] makes possible, makes a major contribution to scalability.

[1] The terms "predicate" and "pattern" are used almost interchangeably; "pattern" usually refers to a unary predicate.

[2] http://www.eclipse.org

[3] http://pmd.sourceforge.net

[4] http://jakarta.apache.org/bcel

For the sake of elegance and brevity of expression, JTL features specific constructs for set manipulation, quantification and other means that eliminate much of the need for loops (recursive calls in the logic programming world). As a result, unlike DATALOG and PROLOG [25] queries, JTL predicates are defined by a single rule, written in a handful of lines, and often in a single line.

Underlying JTL's syntax and semantics is first order predicate logic with no function symbols and augmented with transitive closures, denoted FOPL*. The first order logic represented by JTL is restricted to finite structures (assuming a given database approximation). An inherent difficulty with FOPL** is that it allows one to make cyclic and senseless statements such as "*predicate p holds if and and only if the negation of p holds.*". The language preprocessor therefore restricts queries to to DATALOG with stratified negation [77]. This allows us to be enjoy the theoretical advantages of the formalism, including *modular specification*, *polynomial time complexity*, and a wealth of *query optimization techniques* [39]. Indeed, the JTL compiler under development will generate DATALOG output for an industrial-strength DATALOG engine.

Nevertheless, our experience indicates that stratified negation does not always sufficient expressive power. Currently, the query analyzer of JTL fails to deal with queries. As a result, some of the complex queries are not guaranteed to terminate.

An interesting contribution of this work is in demonstrating that a simple query-by-example like syntax is possible for many tasks of querying OO programs, and in showing that this syntax stands on a solid theoretical ground. It may be possible to put together a JAVA-like syntax for JAVA queries in an ad hoc fashion. The challenge we took upon ourselves was the combination of the sound underlying computational model and the query-by-example front end. Also, in using a DATALOG-based model (and in contrast with PROLOG as some other recent tools do) we achieve a termination guarantee, and the wealth of theory on database query optimization for concrete scalable implementation.

SQL, and more generally, the relational model was sometimes used for software query [72]. However, as Consents, Mendelian and Ryan [22] observed, program analysis frequently requires transitive closure. This is the reason that JTL allows recursion, and is similar in its computational expressive power to Consents et al.ś *Graphing* system.

As the name suggests, JTL is specific to JAVA. In particular, the bountiful class file format of JAVA, its extensive documentation and the verification process, made it possible to carry out the JTL processing on the binary- rather than on the source representation of a program. Such processing of binaries is not possible in languages such as C++ [73]. A port of JTL may therefore need to resort to less efficient and more intricate processing of the source code.

Other issues involved in generalizing JTL ideas to other programming languages and software model representations should become clearer with the more detailed exposition. It should be evident for example, that the bulk of JTL can be easily adapted to C$^\#$ [46]. The most important non-portable part is the dataflow analysis, which cannot be complete in C$^\#$. The reason is that unlike JAVA, the verification process of C$^\#$ is only partial.

Outline Sec. 2 is a brief language tutorial, which shows how JTL can be used to inspect the non-imperative aspects of JAVA code, i.e., everything but the method bodies. Queries of the imperative aspects of the code are the subject of Sec. 3. An explanation of the semantics is then presented in Sec. 4.

Readers who are more interested in actual code may choose to leap directly to Sec. 5, which presents some more advanced applications and code samples.

Sec. 6 compares the performance of JTL with that of JQuery, a JAVA query tool which uses a similar underlying paradigm. Sec. 7 elaborates further on the motivation, surveying existing systems and discusses potential applications. Sec. 8 concludes.

2. The JTL Language

This section gives a brief tutorial of JTL, assuming some basic familiarity with logic programming. The text refers only to the inspection of the non-imperative aspects of the software. The next section will build upon this description the mechanism for delving into method bodies. Readers interested more in the underlying semantics are encouraged to skip forward to Sec. 4, returning here only for reference.

The main issues to note here are the language *syntax*, in which a JAVA program element is matched by a JTL pattern which is very similar in structure to that element (see Sec. 2.1), and the *extensions* to the logic paradigm, specifically, arguments list patterns (Sec. 2.2), transitive closure standard predicates (Sec. 2.3), and quantifiers (Sec. 2.4) which make it possible to achieve many programming tasks without recursion.

The two most important data types, what we call *kinds*, of JTL are (i) **`MEMBER`**, which represents all sorts of class and interface members, including function members, data members, constructors, initializers and static initializers; and (ii) **`TYPE`**, which stands for JAVA **`class`**es, **`interface`**s, and **`enum`**s, as well as JAVA's primitive types such as **`int`**.

Another important kind is **`SCRATCH`**, which, as the name suggests, stands for a temporary value used or generated in the course of computation of a method. Scratches originate from a dataflow analysis of a method, and are discussed below at Sec. 3.

A JTL program is a set of definitions of named logical *predicates*. Execution begins by selecting a predicate to execute as a *query*.

As in PROLOG, predicate names start with a lower-case letter, while *variables* and parameters names are capitalized. Identifiers may contain letters, digits, or an underscore. Additionally, the final characters of an identifier name may be "`+`" (plus), "`*`" (asterisk), or "`'`" (single quote).

2.1 Simple Patterns

Many JAVA keywords are native patterns in JTL, carrying essentially the same semantics. For example, the keyword **`int`** is also a JTL pattern **`int`**, which matches either fields of type **`int`** or methods whose return type is **`int`**. The pattern **`public`** matches all **`public`** program elements, including public class members (e.g., fields) and public classes. Henceforth, our examples shall use these keywords freely; no confusion should arise.

Not all JTL natives are JAVA keywords. A simple example is `anonymous`, defined on **`TYPE`**, which matches anonymous classes.

Some patterns (like **`abstract`**) are overloaded, since they are applicable both to types and members. Others are monomorphic, e.g., **`class`** is applicable only to **`TYPE`**.

*Another example is pattern `type`, defined only on **`TYPE`**, which matches all values of **`TYPE`**. This, and the similar pattern `member` (defined on **`MEMBER`**) can be used to break overloading ambiguity.*

JTL has two kinds of predicates: *native* and *compound*. Native predicates are predicates whose implementation is external to the language. In other words, in order to evaluate native predicates, the JTL processor must use an external software library accessing the code. Native patterns hence are declared (in a pre-loaded configuration file) but not defined by JTL.

In contrast, compound patterns are defined by a JTL expression using logical operators. The pattern

```
public, int                                                    (2.1)
```

matches all **`public`** fields of type **`int`** and all **`public`** methods whose return type is **`int`**. As in PROLOG, conjunction is denoted by a comma. In JTL however, the comma is optional; patterns

separated by whitespace are conjuncted. Thus, (2.1) can also be written as **public int**.

As a matter of style, the JTL code presented henceforth denotes conjunction primarily by whitespace; commas are used mainly for readability—breaking long conjugation sequences into subsequences of related predicates; Disjunction is denoted by a vertical bar, while an exclamation mark stands for logical negation. Thus, the pattern

```
public | protected | private
```

matches JAVA program elements whose visibility is not default, whereas `!public` matches non-`public` elements.

Logical operators obey the usual precedence rules, i.e., negation has the highest priority and disjunction has the lowest. Square parenthesis may be used to override precedence, as in

```
!private [byte|short|int|long]
```

which matches non-`private`, integral-typed fields and methods.

A *pattern definition* names a pattern. After making the following two definitions,

```
integral := byte | short | int | long;
enumerable := boolean | char;
```

the newly defined patterns, `integral` and `enumerable`, can be used anywhere a native pattern can be, as in e.g.,

```
discrete := integral | enumerable
```

Beyond the natives, JTL has a rich set of pre-defined *standard* patterns, including patterns such as `integral`, `enumerable`, `discrete` (as defined above), `method`, `constructor` (both with the obvious semantics), the predicate

```
extendable := !final type
```

(matching classes and interfaces which may have heirs), predicate

```
overridable := !final !static method
```

(methods which may be overridden), and many more.

2.2 Signature Patterns

Signature patterns pertain to (a) the name of classes or members, (b) the type of members, (c) arguments list, (d) declared thrown exceptions, and (e) annotations (meta-data).

Name Patterns A *name pattern* is a regular expression preceded by a single quote, or a previously-declared name. Standard JAVA regular expressions[5] are used, except that the wildcard character is denoted by a question mark rather than a dot. Name literals and regular expressions are quoted with single quotes. The closing quote can be omitted if there is no ambiguity.

For example, `void 'set[A-Z]?*' method` matches any **void** method whose name starts with "`set`" followed by an upper-case letter.

If the name pattern does not contain any regular expression operators, as in

```
toString_p := 'toString method;                (2.2)
```

then the pattern can be made clearer by using a **name** statement to declare `toString` as a member name and get rid of the quote. Thus, an alternative definition of (2.2) is

```
name toString;
toString_p := toString method;                 (2.3)
```

*In truth, the above is redundant, since an implicit **name** statement pre-declares all methods of the JAVA root class `java.lang.Object`.*

Type Patterns *Type patterns* make it possible to specify the JAVA type of a non-primitive class member. A type pattern is a regular expression preceded by a forward slash , e.g., pattern `/java.util.?*/ method` matches all methods with a return type from the `java.util` package or its sub-packages. The closing slash is optional.

[5] as defined by `java.util.regex.Pattern`.

The distinction between type patterns and name patterns only makes sense for members. In matching types, there is no such distinction, and both kinds of literals can be used.

The forward slash is not necessary for type names which were previously declared as such by a **typename** declaration. For example,

```
typename java.io.PrintStream;
printstream_field := PrintStream field;        (2.4)
```

matches any field whose type is `java.io.PrintStream`.

*The **typename** statement in (2.4) declares `java.io.Serializable` as a name of a type, similarly to **name** statement.*

All the types (including classes, interfaces and enumerations) declared in the `java.lang` package are pre-declared as type names, including `Object`, `String`, `Comparable`, and the wrapper classes (`Integer`, `Byte`, `Void`, etc.).

Here is a redefinition of `toString_p` pattern (2.3), which ensures that the matched method returns a `String`.

```
toString_p := String toString method;          (2.5)
```

Arguments List Patterns JTL provides special constructs which all but eliminate recursion. An important example is *arguments list patterns*, used for matching against elements of the list of arguments to a method. (Internally, such lists are stored in a linked list of elements of kind **TYPE**, using standard PROLOG-like `head` and `tail` relations.)

The most simple argument list is the empty list, which matches methods and constructors that accept no arguments. Here is a rewrite of (2.5) using such a list:

```
toString_p := String toString();
```

(Note that the above does not match fields, which have no argument list, nor constructors, which have no return type.)

An asterisk ("`*`") in an arguments list pattern matches a sequence of zero or more types. Thus, the standard pattern

```
invocable := (*);
```

matches members which may take any number of arguments, i.e., constructors and methods, but not fields, initializers, or static initializers. An underscore ("`_`") is a single-type wildcard, and can be used in either the argument list or in the return type. Hence,

```
public _ (_, String, *);                       (2.6)
```

matches any public method that accepts a `String` as its second argument, and returns any type. (Again, aonstructors fail to match (2.6), since they have no return type.)

Other Signature Patterns There are patterns for matching the **throws** clause of the signature, e.g.,

```
io_method := method
        throws /java.io.IOException;
```

There are also patterns which test for the existence or absence of specific annotations in a class, a field or a method, and for annotation values. For example, the following pattern will match all methods that have the `@Override` annotation:

```
@Override method
```

These are not discussed here in detail further in respect of space limitations.

2.3 Variables

It is often useful to examine the program element which is matched by a pattern. JTL employs variable binding, similar to that of PROLOG, for this purpose. For example, by using variable `X` twice, the following pattern makes the requirement that the two arguments of a method are of the same type:

```
firstEq2nd := method (X,X);
```

Similarly, the pattern

```
return_arg := RetType (*, RetType, *);
```

matches any method whose return type is the same as the type of one of its arguments.

Predicates Patterns are parameterless predicates. In general, it is possible to define predicates taking any number of parameters. As usual in logic programming, *parameters* are nothing more than externally accessible variables. Consider for example the predicate

```
is_static[C] := static field _:C;                    (2.7)
```

which takes parameter `C`. When invoked with a specific value for parameter `C`, pattern `is_static` matches only **`static`** fields of that exact type.

Conversely, if the predicate is invoked without setting a specific value for `C`, then it will assign to `C` the types of all **`static`** fields of the class against which it is matched. The semantics by which a parameter to a predicate can be used as *either* input or output is standard in logic programming; the different assignments to `C` are made by the evaluation engine.

Note however that since JTL uses a database-, DATALOG-like semantics, rather than the recursive evaluation engine of PROLOG, each type `C` satisfying (2.7) will show only once in the output, even if there two or more fields of that type.

Native Predicates JTL has several native parameterized predicates. The names of many of these are JAVA keywords. For example, predicate **`implements`**`[I]` holds for all classes which implement directly the parameter `I` (an **`interface`**).

This is the time to note that the predicates **`implements`**`[I]` and `is_static[C]`, just as all other patterns presented so far, have a hidden argument, the *receiver*, also called the *subject* of the pattern, which can be referenced as **`This`** or `#`.

Other native predicates of JTL include **`members`**`[M]` (true when `M` is one of **`This`**'s members, either inherited or defined), **`defines`**`[M]` (true when `M` is defined by **`This`**), **`overriding`**`[M]` (true when **`This`** is a method which overrides `M`), `inner[C]` (true when `C` is an inner class of **`This`**), and many more.

The following example shows how a reference to **`This`** is used to define a pattern matching what is known in C++ jargon as "copy constructors":

```
copy_ctor := constructor(T), T.members[This];
```

(2.8)

This example also shows how a predicate can be applied to a subject which is not the default, by using a JAVA-like dot notation.

The `copy_ctor` *predicate works like this: first, the pattern* `constructor(T)` *requires that the matched item, i.e.,* **`This`**, *is a constructor, which accepts a single parameter of some type* `T`. *Next,* `T.`**`members`**`[`**`This`**`]` *requires that* **`This`***—the matched constructor—is a member of its argument type* `T`, *or in other words, that the constructor's accepted type is the very type that defines it.*

Literals, just as variables, can be used as actual parameters. For example, **`class implements`**`[M]` matches any class that implements interface `M`, whereas

```
interface extends[/java.io.Serializable]
```

matches any interface that extends the `Serializable` interface.

The square brackets in an invocation of a predicate with a single parameter are optional. The above could thus have also been written as:

```
interface extends /java.io.Serializable
```

Moreover, since there is a clear lexical distinction between parameters and predicates, even the dot notation is not essential for changing the default receiver. Thus,

```
copy_ctor := constructor(T), T members This;
```

is equivalent to (2.8).

Standard Predicates JTL also has a library of standard predicates, many of which are defined as transitive closure of the native predicates. Fig. 2.1 shows a sample of these.

Figure 2.1 Some of the standard predicates of JTL

```
inherits[M] := members[M] !defines[M];
container[C] := C.members[This];
precursor[M] := M.overriding[This];
implementing[M] := !abstract,
      overriding[M] M.abstract;
abstracting[M] := abstract,
      overriding[M] !M.abstract;
extends+[C] := extends[C] |
      extends[C'] C'.extends+[C];
extends*[C] := C is This | extends+[C];
interfaceof[C] := C.class C.implements[This];
interfaceof+[C] := C.implements+[This];
interfaceof*[C] := C.implements*[This];
```

The figure makes apparent the JTL naming convention by which the reflexive transitive closure of a predicate `p` is named `p*`, while the anti-reflexive variant is named `p+`. The myriad of recursive definitions such as these saves much of the user's work; in particular it is rare that the programmer is required to employ recursion.

It is interesting to examine the "recursive" definition of one of these predicates, e.g., **`extends+`**:

```
extends[C] | extends[C'] C'.extends+[C]
```

It may appear at first that with the absence of a halting condition, the recursion will never terminate. A moment's thought reveals that this is not the case. Since JTL uses a bottom-up construction of facts, starting at a fixed database, the semantics of this recursive definition is not of stack-based recursive calls, but rather as dynamic programming, or work-list, approach for generating facts.

Predicate Name Aliases The name **`extends+`** suggests that it is used as a verb connecting two nouns. As mentioned above, we can even write

```
C extends+ C'
```

But, the same predicate can be used in situations in which, given a class `C`, we want to generate the set of *all* classes that it extends. A more appropriate name for these situations is `ancestors`. It is possible to make another definition

```
ancestors[C] := extends+[C];
```

To promote meaningful predicate names, JTL offers what is known as *predicate name aliases*, by which the same predicate definition can introduce more than one name to the predicate. Aliases are written as an *definition annotation* which follows the main rule The definition of **`extends+`** has such an alias

```
extends+[C] := extends C |
        extends C', C'.extends+[C];
    Alias ancestors;
```

The use for an alias named `ancestors` will become clear with the presentation of predicate (2.10) below.

Native predicates can also have aliases, which are specified along with their declaration.

2.4 Queries

As mentioned previously, JTL's expressive power is that of FOPL*. Although it is possible to express universal and existential quantification with the constructs of logic programming, we found that the alternative presented in this section is more natural for the particular application domain.

Consider for example the task of checking whether a JAVA class has an **`int`** field. A straightforward, declarative way of doing that is to examine the set of all of the class fields, and then check whether this set has a field whose type is **`int`**.

The following pattern does precisely this, by employing a *query* mechanism:

```
has_int_field := class members: {
                    exists int field;          (2.9)
                 };
```

Here, the query **members**: { $Q_1; \cdots ; Q_n$ } generates first the set of all possible members M, such that #.**members**[M] holds. (The "**members**:" portion of the query is called the *generator*.)

This set is then passed to Q_1 through Q_n, the *quantifiers* embedded in the curly brackets. The entire query holds if all of these quantifiers hold for this set.

In (2.9), there was only one quantifier: the JTL statement **exists int** field is an existential quantifier which holds whenever the given set has an element which matches the pattern **int** field.

The next example shows two other kinds of quantifiers.

```
class ancestors: {
    all public;
    no abstract;                               (2.10)
};
```

The evaluation of this pattern starts by computing the generator. In this case, the generator generates the set of all classes that the receiver **extends** directly or indirectly, i.e., all types C for which #.ancestors[C] holds. (Recall that ancestors is an alias for **extends+**, defined above.) The first quantifier checks whether all members of this set are **public**. The second quantifier succeeds only if this set contains no **abstract** classes. Thus, (2.10) matches classes whose superclasses are all public and concrete.

Quantifiers in JTL include also, **many** p holds if the queried set has two or more elements for which pattern p holds; whereas **one** p holds if this set has precisely one such element.

The existential quantifier is the most common; hence the **exists** is optional. Also, a missing generator (in predicates whose subject is a **TYPE**) defaults to the **members**: generator. Hence, a concise rewrite of (2.9) is

```
has_int_field := class {
                    int field;                 (2.11)
                 };
```

In the two examples shown here, the generator was a predicate with a single named parameter and an implicit receiver. In such cases, the generator generates a set of primitive values, which are the possible assignments to the argument. However, in general, the generator generates a relation of named tuples, and the quantifiers are applied to the set of these tuples. We discuss the underlying semantics of queries in greater detail in Sec. 4.

3. Queries of Imperative Code

The executional aspect of JAVA code remained beyond the description of JTL in the previous section. This aspect is primarily method bodies, but also other imperative code, including constructors, field initializers and static initializers.

Now that the bulk of the language syntax is described, we can turn to the question of queries of *imperative entities*. To an extent, queries of these entities are mostly a matter of library design rather than a language design. Recall that JTL native predicates are implemented as part of the supporting library that the JTL processor uses for inspecting JAVA code. Extending this library, without changing the JTL syntax, can increase the search capabilities of the language.

Sec. 3.1 shows how by adding a set of native predicates, JTL *can* be extended to explore an abstract syntax tree representation of the code. This section also explains why this extension was not implemented yet. We chose instead to implement a pedestrian set of natives that make it possible to explore the fields and methods that executional code uses, as described in Sec. 3.2. The more sophisticated mechanism that JTL uses for inspecting method bodies is through a dataflow analysis, as described in Sec. 3.3.

3.1 Abstract Syntax Trees and JTL

Executional code can be represented by an abstract grammar, with non-terminal symbols for compound statement such as **if** and **while**, for operations such as type conversion, etc. One could even think of several different such grammars, each focusing on a different perspective of the code.

Code can be represented by an abstract syntax tree whose structure is governed by the abstract grammar. To let JTL support such a representation, we can add a new kind, **NODE**, and a host of native relations which represent the tree structure. For example, a native binary predicate **if** can be used to select **if** statement nodes and the condition associated with it; a binary predicate **then** can select the node of the statement to be executed if the **if** condition holds; another binary predicate, **else**, may select the node of the statement of the other branch, etc.

As an example of an application for such a representation, consider a search for cumbersome code fragments such as

```
if (C)
   return true;
else
   return false;
```

with the purpose of recommending to the programmer to write

```
   return C;
```

instead. The following pattern matches such code:

```
boolean_return_recommendation :=
   if[_] then[S1] else[S2],
      S1.return[V1], S2.return[V2],
        V1.literal["true"],
           V2.literal["false"];
```

The above pattern should be very readable: we see that its receiver must be a **NODE** which is an **if** statement, with a don't-care condition (i.e., _), which branches control to statements S1 and S2; also both S1 and S2 must be **return** statements, returning nodes V1 and V2 respectively. Moreover, the patterns requires that nodes V1 and V2 are literal nodes, the first being the JAVA **true** literal, the second a **false**.

In principle, such a representation can even simultaneously support more than one abstract grammar. Two main reasons stood behind our decision not to implement, or even define (at this stage), the set of native patterns required for letting JTL explore such a representation of the code.

1. *Size.* Abstract grammars of JAVA (just as any other non-toy language) tend to be very large, with tens and hundreds of non-terminal symbols and rules. Each rule, and each non-terminal symbol, requires a native definition, typically more than one. The effort in defining each of these is by no means meager.
2. *Utility.* Clearly, an AST representation can be used for representing the non-imperative aspects of the code. The experience gained in using the non-AST based representation of JTL for exploring these aspects, including type signatures, declaration modifiers, and the interrelations between classes, members and packages, indicated that the abstraction level offered by an abstract syntax tree is a bit too low at times.

A third, (and less crucial) reason is that it is not easy (though not infeasible) to elicit the AST from the class file, the data format used in our current implementation.

3.2 Pedestrian Code Queries

In studying a given class, it is useful to know which methods use which fields. The following JTL pattern, for example, implements

one of Eclipse's warning situations, in which a private member is never used.

```
unused_private_member := private field,
   This is F,
   declared_in C, C inners*: {
      all !access[F];
   }
```

The pattern fetches the class `C` that defines the field, and then uses the reflexive and transitive closure of the `inner` relation, to examine `C`, its inner classes, their inner classes, etc., to make sure that none of these reads or writes to this field. (The unification (**`This is`** `F`) is for making the receiver field accessible inside the curly brackets.)

The pattern `access` showing in the penultimate line of the example is defined in the JTL library. The definition, along with some of the other standard patterns that can be used in JTL for what we call *pedestrian code queries* is shown in Fig. 3.1. Such queries model the method body as an unordered set of byte-code instructions, checking whether this set has certain instructions in it.

Figure 3.1 Standard predicates for pedestrian code queries

```
access[F] := read F | write F; Alias accesses;
read[F] := offers M, M read F; Alias reads;
write[F] := offers M, M read F; Alias writes;

calls[M] := invokes_interface[M] |
      invokes_virtual[M] |
         invokes_static[M] |
            invokes_special[M];
   Alias invokes, invoke;

use[X] := access[M] | invoke[M]; Alias uses;
```

In the figure we see that the definition of `access` is based on the overloaded predicates `read` and `write`. The native predicate `read[F]` (respectively `write[F]`) holds if the receiver is a method whose code reads (respectively writes to) the field `F`. The second (respectively the third) line of the figure, overloads the native definition of `read` (respectively `write`), so that it applies also to receivers whose kind is **`TYPE`**.

The figure also makes uses of the four other pedestrian natives for inspecting code: `invokes_interface`, `invokes_virtual`, `invokes_static`, and `invokes_special`. (These natives also have aliases identical to the bytecode mnemonics.)

With this minimal set of six natives, several interesting patterns can be defined. For example, predicate

```
creates[T] := invokes_static[M], M.ctor,
             M.declared_in[T];
```

is true when the receiver creates an object of type `T`. Also, the following predicates test whether a method *refines* its precursor

```
refines[M] := overrides[M] invokes_special[M];
refines := refines[_]; Alias refiner;
```

The following predicate checks whether a method is not empty

```
does_something := !void | invokes[_] |
   writes[_];
```

(If a method does not return a value, does not invoke any other method, nor write to a field, then it must have no meaningful effect.) With the above, we implemented an interesting PMD rule, signalling an unnecessary constructor, i.e., the case that there is only one constructor, it is public, has an empty body, and takes no arguments.

```
unnecessary_constructor := class {
   constructor => public () !does_something;
}
```

The following predicate identifies a case that a constructor calls another constructor `C` of class `T`.

```
c_call[C,T] := constructor invokes_special[C]
    C.constructor C.declared_in[T]
```

With this predicate, we can present three rules which identify the different ways that a constructor may begin its mission in JAVA.

```
c_delegation[C] := // First line is ''this(...)''
   declared_in[MyClass] c_call[C,MyClass];
c_refinement[C] := // First line is ''super(...)''
   declared_in[MyClass] c_call[C,Parent],
   MyClass.extends[Parent];
c_handover[C] := // Either
   ctor_delegation[C] | ctor_refinement[C];
```

3.3 Dataflow Code Queries

As a substitute to AST queries and at a higher level of abstraction than the pedestrian queries, stand dataflow code queries. In the course of execution of imperative entities many temporary values are generated. Dataflow analysis studies the ways that these values are generated and transferred. The idea is similar to dataflow analysis as carried out by an optimizing compiler [2, Sects. 10.5–10.6], or by the JAVA bytecode verifier [55, Sec. 4.92].

3.3.1 Scratches for Dataflow Analysis

To implement dataflow analysis, we introduce a new JTL kind, **`SCRATCH`**, which represents what is called in the compiler lingo a "variable definition", i.e., an assignment to a temporary variable. In JAVA we find two categories of temporary variables on what is called a "method frame":

1. *A location in the "local variables array",* including the locations reserved in this array for method parameters, and in particular the "**`this`**" parameter of instance methods, as well as local variables that an imperative entity may define.[6]
2. *A location in the "operands stack",* used for storing temporary variables in the course of the computation.

As usual with dataflow analysis, there is a fresh scratch for each assignment to a temporary variable. Also, scratches are generated on a merge of control flow. A scratch is anonymous; it does not carry with it its location in the frame.

An assignment to a scratch can be from one of the following sources: another scratch, an input parameter value, a constant, a field, a value returned from a method or a code entity, an arithmetical operation, or a thrown exception. A scratch can be assigned to another scratch, passed as a parameter, assigned to a field, thrown or returned.

The dataflow analysis is implemented at the class file level. Obviously, this can only be done with a non-optimizing compiler, which makes a lossless translation of the source dataflow into an equivalent dataflow of the intermediate or target language. Luckily, the standard JAVA compiler obeys this requirement.

It should be clear that the dataflow information and analysis we carry is also feasible by starting from the source level, generating temporaries for all intermediate values.

On the other hand, it should be noted that JAVA semantics and tradition also helped in making our dataflow analysis more effective; among the contributing factors we can mention the tendency to use short methods, the requirement that all locals are initialized before they are uses, call-by-value semantics, no pointer arithmetic, our decision to ignore arrays, etc.

[6] Note that the verification process guarantees that we can treat two adjacent locations which are used for storing a **`long`** or **`double`** variables.

3.3.2 Using Scratches

The following pattern, capturing the authors' understanding of the notion of *setter*, gives a quick taste of the manner of using data flow information in JTL.

```
setter := void instance method(_) !calls[_] {
  putfield[_,_] => parameter;
  one putfield[_,_];
  putfield[_,Ref] Ref.this.
  no [ putstatic[_] | get[_] | compared ];
};
```

The first line in the above requires that the receiver is a **void** *instance method, taking a single parameter, and (by using a pedestrian predicate) that it calls no other methods. The predicates in the curly brackets make the following requirements in order: (i) all assignments to a field are of a scratch that is provably equal to the single parameter of the method; (ii) there is precisely one assignment to an instance field in the method; (iii) this assignment is to a field using an object reference which is provably equal to the* **this** *implicit parameter of the method; and that (iv) there are no assignments to a static field, nor field retrievals, nor a comparison in the method.*

The example shows that the formulation of data flow requirements is not so simple. Moreover, the precise notion of a setter is debateable. We argue that JTL's terse, English-like syntax, augmented with the natural coding of work-list algorithms in the logic paradigm, help in quick testing and experimenting with patterns such as the above.

Dataflow analysis is a large topic, and its application in JTL involves about three dozens of predicates. We can only give here a touch of the structure of JTL's predicates library and the patterns that can be written with it.

Tab. 1 lists the essential native unary predicates defined on scratches. The predicates are obtained by a standard (conservative) dataflow analysis similar to that of the verifier does. Thus, predicate `parameter` holds for all scratches which are provably equal to a parameter, `nonnull` for temporaries which are provably non-null, etc.

Table 1. Native unary predicates of scratches

Predicate	Meaning
`parameter`	a method parameter stored in the LVA
`constant`	a constant
`null`	the null constant
`nonnull`	cannot be the null constant
`temp`	an operand-stack scratch
`this`	equal to parameter 0 of an instance method
`local`	an (uninitialized) automatic variable in the LVA
`returned`	returned by the code
`athrow`	thrown by the code
`caught`	obtained by catching an exception
`compared`	compared in the code

The native binary predicate **`scratches`**`[S]` (also aliased as `has_scratch`) holds if `S` is a scratch of the method **`This`**, and serves as the default generator of methods. Hence, the curly brackets in the pattern

```
instance method {
   returned => this;
}
```

iterate over all scratches of a given method, checking that every scratch returned by the method is equal to the **`this`** parameter. Also, the binary predicate `typed[T]` holds if `T` is the type of the scratch `#` (**`This`**). The following pattern returns the set of all types that a method uses:

```
use_types[T] := method has_scratch[S]
                  S.typed[T];
```

The most important predicate connecting scratches is `from[S]`, which holds if scratch `S` is assigned to scratch `#`. Similarly, `func[S]` holds if `#` is computed by an arithmetical computation. We also have the definition

```
depend[S] := func[S] | from[S];
```

As usual, `from*`, `func*` and `depend*` denote the reflexive transitive closure of `from`, `func` and `depend`.

There is also a native predicate for each of the four bytecode instructions used for accessing fields. For example, the binary predicate `putstatic[F]` holds if the scratch `#` is assigned to **`static`** field `F`, while `getfield[F,S]` holds if `#` is retrieved from field `F` with scratch `S` serving as object reference.

In addition, there are two predicates for each of the instructions for method calling, e.g., a predicate `invokespecial[M,S]` holds if scratch `#` is used as an argument to a call of method `M`, where scratch `S` serves as the receiver, and a predicate `get_invokestatic[M]` which holds if `#` obtains its value from a static call to method `M`.

For a given scratch there is typically more than one `S`, such that `from[S]`, or `from*[S]` holds [7]. Dealing with this multiplicity of dataflow analysis is natural to DATALOG programming. For example, pattern `dead` identifies dead scratches, i.e., scratches whose values are never used:

```
unassigned := !put[_] ! [ _.from[#] |
        _.func[#] | ...etc. ];
dead := !compared unassigned ;
```

The following predicate selects all sources of values that may be assigned to a scratch:

```
origins[S] := from*[S],
    [ S.parameter | S.constant | S.get[_] ];
```

In words, `origins[S]` holds if `S` is a port of entry of an external value which may eventually be assigned to `#`. We can now write a pattern which determines whether a method returns a constant.

```
constant_method := method {
   one returned;
   returned => origins: { all constant; }
}
```

Using an overloaded version of `from[X]` which holds if `#` obtains its value either from a field `X` or from a call to method `X`, it is mundane to extend the above pattern to match also method returning a **`final`** field. It should also be obvious that dataflow analysis provides enough information so that the implementation of pattern capturing, say, a getter notion, or even the selection of fields, is not too difficult.

The native predicate `locus[S]` holds if `S` and `#` are distinct scratches which are stored in the same location on the frame. This predicate is used in Fig. 3.2, together with `dead` and `unassigned` in a bunch of predicates which implement several PMD advices. .

Figure 3.2 Implementation of some PMD warnings with scratches

```
changed_parameter := parameter locus[_];
multiple_returns := method returns: { many; };
null_assignment := from[C], C.null;
unread_parameter := dead parameter;
flag_parameter := unassigned compared parameter;
unassigned_local := local locus: { empty };
unused_local := dead local;
```

[7] other than in linear methods, i.e., methods whose control flow contains no branch statements

4. Underlying Semantics

As stated above, JTL belongs to the logic programming paradigm. This section explains how the JTL constructs are mapped to familiar notions of the paradigm.

In a nutshell, JTL is a *simply typed* formalism whose underlying semantics is *first order predicate logic* augmented with *transitive closure* (FOPL*). Evaluation in JTL is similar to that of PROLOG (more precisely, DATALOG), with its built-in support for the "join" and "project" operations of relational databases. This section elaborates the language semantics a bit further.

Kinds and Predicates The type system of JTL consists of a fixed finite set of primitive kinds (types) $\mathcal{T}$. There are no compound kinds.

A *predicate* is a boolean function of $T_1 \times \cdots \times T_n$, $n \geq 0$, where $T_i \in \mathcal{T}$ for $i = 1, \ldots, n$. A predicate can also be thought of as a *relation*, i.e., a *subset* of the cartesian product

$$T_1 \times \cdots \times T_n,$$

called the *domain* of the predicate. By convention, the first argument of a predicate is unnamed, while all other arguments are named. The unnamed arguments is called the *receiver* or the *subject*.

Native Predicates JTL has a number of native predicates, such as **class**—a unary predicate of **TYPE**, i.e., **class** $\subseteq$ **TYPE**, **synchronized** $\subseteq$ **MEMBER** (the predicate which holds only for synchronized methods), **members** $\subseteq$ **TYPE** $\times$ **MEMBER**, **extends** $\subseteq$ **TYPE** $\times$ **TYPE** (with the obvious semantics), and the *0-ary* predicates **false** (an empty 0-ary relation) and **true** (a 0-ary relation consisting of a single empty tuple). Built-in predicates are called in certain communities *Extensional Database* (EDB) predicates.

A run of the JTL processor starts by loading a declaration of arity and argument types of all native predicates from a configuration file. Native declarations are nothing more than definitions without body. For example, the following commands in a configuration file

```
MEMBER.int;
```

states that **int** is a unary predicate such that **int** $\subseteq$ **MEMBER**.

Compound Predicates Conjunction, disjunction and negation can be used to define *compound* predicates from the built-ins. Also permitted, are quantification as explained in Sec. 2 and transitive closure, i.e., recursion as in

```
extends+[X] := extends X |
   extends[Y] Y.extends+[X];
```

The language offers an extensive library of *pre-defined* of *standard* compound predicates. Compound predicates are sometimes called *Intensional Database* (IDB) predicates.

Finite Databases To run, a JTL program requires a database which *conforms* to the natives, i.e., it must have in its schema the relations or the EDBs as dictated by the set of natives defined by the JTL implementation at hand.

The simplest way to supply a database is by specifying to the JTL processor a finite set of classes and methods, e.g., a "`.jar`" file. Obviously, such a collection does not directly represent any EDBs. EDBs are realized on top of the collection by means of a bytecode analysis library.

Alternatively, a finite database can also be provided by supplying a finite set of legal JAVA source files. The native relations are then realized on top of these by a JAVA parser.

JTL queries can also be run without a fixed input set. Such a situation, can be thought of as a DATALOG query over an infinite database.

Evaluation Order Unlike PROLOG, the order of evaluation in JTL is unimportant. The output set of a pattern is the same regardless of the order by which its constituent predicates in it are invoked. Predicates have no side-effects, and all computations (on finite databases) terminate.

The simplest way to compute the output set is bottom-up, i.e., by using a work-list algorithm which uses the program rules to compute all tuples in all IDBs used by the program goal. This process, although guaranteed to terminate, can be very time- and space-inefficient. JTL instead analyzes queries and applies, whenever possible, a more efficient top-down evaluation strategy

Overloading and Kind Inference The JTL processor includes a *kind inference engine* which, based on the kind of arguments and arity of the native predicates, infers arity and arguments kinds of predicates defined on top of these. For example, the definition

```
real := double | float;                    (4.1)
```

implies that `real` $\subseteq$ **MEMBER**.

JAVA's overloading of keywords carries through to JTL, e.g., since the JAVA keyword **final** can be applied to classes and members, the built-in predicate **final** in JTL is overloaded, denoting two distinct relations: $\textbf{final}_1 \subseteq$ **TYPE** and $\textbf{final}_2 \subseteq$ **MEMBER**. Many native predicates are similarly overloaded; JTL infers overloading of compound predicates. For example, the conjunction of **final** and **public** is overloaded; the conjunction of **final** and **interface** is not.

The Default Receiver As seen in the last examples, JTL sports an implicit mechanism of applying a predicate to receiver. For example, the above definition of `real` could have been written as

```
real := #.double | #.float;                (4.2)
```

Named Arguments The *signature* of a relation is an ordered pair $\langle R, \mathbf{A}\rangle$, whose first component, $R \in \mathcal{T}$, defines the type of the receiver, while the second component, $\mathbf{A} = \{\langle \ell_1, A_1\rangle, \ldots \langle \ell_m, A_m\rangle\}$, defines the names of the arguments (the labels ℓ_j, $j = 1, \ldots, m$, must be distinct) and their types ($A_j \in \mathcal{T}$ for $j = 1, \ldots, m$).

A row of a relation is in general a *named tuple*, i.e., a tuple of values, where all but the first carrying labels, such that the types of these values and the labels they carry match exactly the signature of the predicate.

Predicates are characterized by signature, e.g., the signature of predicate **members** is $\big\langle$**TYPE**, $\{\langle$"M", **MEMBER**$\rangle\}$ $\big\rangle$, while the definition

```
container[C] := C.members[#]
```

implies

$\big\langle$**MEMBER**, $\{\langle$"C", **TYPE**$\rangle\}$ $\big\rangle$

as the signature of `container`. Overloaded predicates have multiple signatures, one for each meaning. For example, the built-in predicate **final** has two signatures, $\big\langle$**MEMBER**, $\emptyset\big\rangle$ and $\big\langle$**TYPE**, $\emptyset\big\rangle$.

Baggage In addition to the receiver and to the named parameters, every JTL predicate has another hidden output parameter. The term *baggage* is used for this parameter since its computation is done behind the scenes, without programmer intervention, and further, in many cases it is eliminated by the JTL processor.

The baggage parameter is always of kind **STRING**, and it defaults, in most predicates, to the predicate name. So, the returned baggage of the predicate **public** is simply the string `"public"`, the baggage of a regular expression matching the name of a field or a method is the name of that class member.

Also, by default, the baggage a conjunction is the concatenation of the components' baggage, in the order these component appear. The returned baggage of predicate **public int** is therefore

the string "public int". The default baggage of most other constructs for making compound predicates is simply the empty string.

Although in principle, a programmer may override the baggage value, and even check it, current support for doing so in JTL is minimal.

Set Expressions JTL extends logic programming with what we call a *query*, which is a predicate whose evaluation involves the generation of a temporary relation, and then applying various *set expressions* (e.g., quantifiers) to this relation. A query predicate is true if all the set conditions hold for the generated temporary relation.

The predicate defined in Fig. 4.1 tries to check that a class is "classical", i.e., that it has at least one field, two or more methods, that all methods are public, all fields are private, that there are no static fields or methods, and that the sets of "setters" and "getters" of this class are disjoint.

Figure 4.1 A JTL predicate for matching "classical class" notion.

```
classical := class members: {
    has field;
    many method;
    no static;
    method => public;
    field => private;
    disjoint setter, getter;
}
```

(The definition in the figure assumes that predicates `setter` and `getter` were previously defined.)

The essence of the example is the *generator* of the temporary relation, written as **members:**. The colon character (`:`) appended to predicate **members** makes it into a generator. JTL generates the set of all members `M`, such that `#.`**members**`[M]` holds. This set, which can be also thought of as a relation with only one unnamed column, is subjected to the set expressions inside the curly brackets.

Six conditions are applied to this set: the first is an existential quantifier (**has** is synonymous to **exists**) requiring that at least one element in the generated set satisfies the `field` condition, i.e., that the class has at least one field. The second condition similarly requires that `method` holds to two more members. The 3rd condition, as should be obvious, requires that this set does not contain any static members.

The 4th condition is a *set expression* requiring that the predicate `method` *implies* **public** holds in this set, i.e., that method members are public. The 5th condition similarly states that the set of `field` members is a subset of the set of **private** members. Finally, the set expression **disjoint** `setter, getter` requires that the two subsets obtained by applying predicates `setter` and `getter` to the set of class members are disjoint.

5. Applications

Having presented the JTL syntax, the language's capabilities and its underlying semantics, we are in a good position to describe some of the applications.

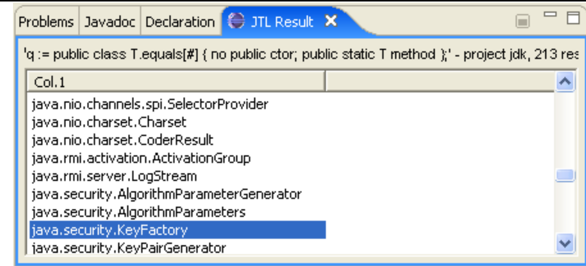
5.1 Integration in CASE Tools and IDEs

In their work on JQuery, Janzen and De Volder [50] make a strong case, including empirical evidence, for the need of a good software query tool as part of the development environment.

As detailed below in Sec. 7.1 and demonstrated by Tab. 2, the querying (but not the navigational) side of JQuery can be replaced and simplified by JTL.

We have developed an Eclipse plug-in that runs JTL queries and presents the result in a dedicated view. Fig. 5.1 shows an example: the program (which appears above the results) found all classes in JAVA's standard library for which instances are obtained using a **static** method rather than a constructor. Using JTL,

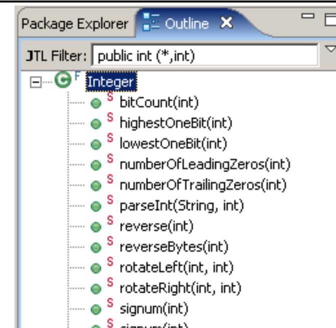
Figure 5.1 Screenshot of the result view of JTL's Eclipse plugin



many searches can be described intuitively. For example, to find all classes that share a certain annotation `@X`, the developer simply searches for `@X` **class**. The similarity between JTL syntax and JAVA declarations will allow even developers who are new to JTL to easily and effectively sift through the overwhelming number of classes and class members in the various JAVA libraries.

JTL can also be used to replace the hard-coded filtering mechanism found in many IDEs (e.g., a button for showing only **public** members of a class) with a free-form filter. Fig. 5.2 is a mock screenshot that shows how JTL can be used for filtering in Eclipse.

Figure 5.2 Using JTL for filtering class members (mock)



Finally, JTL can be used for search-and-replace operations. Since the operation is context-sensitive, there is no risk of, e.g., changing text that appears in comments. With the current version of JTL, this is limited to changing class and method signatures, and is therefore less powerful than Eclipse's built-in refactoring facilities. However, unlike these facilities, the changes that JTL allows are open-ended and not limited to a pre-defined set of operations.

For example, the following JTL program will make every method called `getLock()` synchronized, without changing any other part of the method's signature:

```
!synchronized getLock()
   [* synchronized #sig *]
```

Currently, JTL has only rudimentary support for program transformation. In particular, as explained above, every JTL predicate returns a baggage string which can be used for producing output. This returned value, may be used to generate a replacement program fragment, although JTL does not offer still means for ensuring that it is indeed syntactically valid string.

By default, this returned string is the predicate name. As shown above, the construct `[* S *]`*, where* `S` *is a string expression can be used to change this default. In the example the returned value is the string* **synchronized** `#sig`*, where* `sig` *is a library predicate that returns the member's signature as a valid* JAVA *source fragment.*

5.2 Specifying Pointcuts in AOP

The limited expressive power of the pointcut specification language of ASPECTJ (and other related AOP languages, e.g., CAESAR [59] and ASPECTJ2EE [21]), has been noted several times in the literature [42, 64].

We propose that JTL is integrated into AOP processors, taking charge of pointcut specification. To see the benefits of using a JTL component for this purpose, consider the following ASPECTJ pointcut specification:

```
call(public void *.set*(*));
```

JTL's full regular expressions syntax can be used instead, by first defining

```
setter := public void 'set[A-Z]?*'(_);        (5.1)
```

and then writing `call(setter)`. Unlike the ASPECTJ version, (5.1) uses a proper regular expression, and therefore does not erroneously match a method whose name is, e.g., `settle()`.

Fig. 5.3 presents an array of ASPECTJ pointcuts trapping read and write operations of primitive public fields. Not only tedious, it is also error prone, since a major part of the code is replicated across all definitions.

Figure 5.3 An ASPECTJ pointcut definition for all read- and write-access operations of primitive public fields.

```
get(public boolean *) || set(public boolean *) ||
get(public byte *) || set(public byte *) ||
get(public char *) || set(public char *) ||
get(public double *) || set(public double *) ||
get(public float *) || set(public float *) ||
get(public int *) || set(public int *) ||
get(public long *) || set(public long *) ||
get(public short *) || set(public short *);
```

By using disjunction in JTL expressions, the ASPECTJ code from Fig. 5.3 can be greatly simplified if we allow pointcuts to include JTL expressions:

```
primitive := boolean | byte | char | double |
      float | int | long | short;
ppf := public primitive field;

get(ppf) || set(ppf); // JTL−based AspectJ pointcut
```

The ability to name predicates, specifically `ppf` in the example, makes it possible to turn the actual pointcut definition into a concise, readable statement.

The following is an example of a condition that is impossible to specify in ASPECTJ:

```
setter := public void 'set[A-Z]?*'(_);
boolean_getter = boolean 'is[A-Z]?*'();
other_getter = !boolean !void 'get[A-Z]?*'();
getter := public
      [boolean_getter | other_getter];

field_in_plain_class := public field,
   declare_in[C], C.members: {
      no getter;
      no setter;
   };
```

Condition `field_in_plain_class` holds for **`public`** fields in a class which has no getters or setters. This requirement is realized by predicate `container`, which captures in `C` the container class. A query is then used to examine the other members of the class.

The above could have been implemented in other extensions of the ASPECTJ pointcut specification language, but not without a loop or a recursive call.

Our contribution puts the expressive power of JTL at the disposal of ASPECTJ and other aspect languages, replacing the sometimes ad-hoc pointcut definition language with JTL's systematic approach. There are two limitations in doing that: First, JTL can only be used to make queries on the program static structure, and not on the *dynamic* control flow. The second limitation is more technical: although JTL queries can be easily made from an JAVA program, there is no sufficient API for the client code to intervene in the parsing process, pass error messages, etc.

5.3 Concepts for Generic Programming

In the context of generic programming, a *concept* is a set of constraints which a given set of types must fulfil in order to be used by a generic module. As a simple example, consider the following C++ template:

```
template<typename T>
class ElementPrinter {
public:
   void print(T element) {
      element.print();
   }
}
```

The template assumes that the provided type parameter `T` has a method called `print` which accepts no parameters. Viewing `T` as a single-type *concept* [31, 74], we say that the template presents an implicit assumption regarding the concept it accepts as a parameter. Implicit concepts, however, present many problems, including hurdles for separate compilation, error messages that Stroustrup et al. term "of spectacular length and obscurity" [74], and more.

With Java generics, one would have to define a new interface

```
interface Printable { void print(); };
```

and use it to confine the type parameter. While the concept is now explicit, this approach suffers from two limitations: first, due to the nominal subtyping of JAVA, generic parameters must explicitly implement interface `Printable`; and second, the interface places a "baggage" constraint on the return type of `print`, a constraint which is not required by the generic type.

Using JTL, we can express the concept explicitly and without needless complications, thus:

```
(class | interface) { print(); };
```

There are several advantages for doing that: First, the underlying syntax, semantics and evaluation engine are simple and need not be re-invented. Second, the JTL syntax makes it possible to make useful definitions currently not possible with JAVA standard generics and many of its extensions.

The problem of expressing concepts is more thorny when multiple types are involved. A recent work [31] evaluated genericity support in 6 different programming languages (including JAVA, C# and EIFFEL [49]) with respect to a large scale, industrial strength, generic graph algorithm library, reaching the conclusion that the lack of proper support for multi-type concepts resulted in awkward designs, poor maintainability, and unnecessary run-time checks.

JTL predicates can be used to express multi-type concepts, and in particular each of the concepts that the authors identified in this graph library.

As an example, consider the `memory_pool` concept, which is part of the challenging example the concepts treatise used by Garcia et al. A memory pool is used when a program needs to use several objects of a certain type, but it is required that the number of instantiated objects will be minimal. In a typical implementation, the memory pool object will maintain a cache of unused instances. When an object is requested from the pool, the pool will return a previously cached instance. Only if the cache is empty, a new object is created by issuing a create request on an appropriate factory object.

More formally, the memory pool concept presented in Fig. 5.4 takes three parameters: `E` (the type of elements which comprise the pool), `F` (the factory type used for the creation of new elements), and **`This`** (the pool type).

Figure 5.4 The `memory_pool` concept

```
name create, instance, acquire, release;

factory[E] := (class | interface) {
  public constructor ();
  public E create();
};

memory_pool[F,E] := equals[T] {
  public static T instance();
  public E acquire();
  public release(E);
}, F.factory[E];
```

The body of the concept requires that **`This`** will provide `acquire()` and `release()` methods for the allocation and deallocation (respectively) of `E` objects, and a static `instance()` method to allow client code to gain access to a shared instance of the pool. Finally, it requires (by invoking the `Factory` predicate) that `F` provides a constructor with no arguments, and a `create()` method that returns objects of type `E`.

As shown by Garcia et al., the requirements presented in Fig. 5.4 have no straightforward representation in JAVA, C# or EIFFEL. In particular, using an **`interface`** to express a concept presents extraneous limitations, such as imposing a return type on `release`, and it cannot express other requirements, such as the need for a zero-arguments constructor in a factory. Using an **`interface`** also limits the applicable types to those that `implement` it, whereas the concept itself places no such requirement.

In a language where JTL concept specifications are supported, a generic module parameterized by types `X`, `Y` and `Z` can declare, as part of its signature, that `X.memory_pool[Y,Z]` must hold. This will ensure, at compile-time, that `X` is a memory pool of `Z` elements, using a factory of type `Y`[8].

Concepts are not limited to templates and generic types. Mixins, too, sometimes have to present requirements to their type parameter. The famous `Undo` mixin example [5] requires a class that defines two methods, `setText` and `getText`, but does not define an `undo` method. The last requirement is particularly important, since it is used to prevent *accidental overloading*. However, it cannot be expressed using JAVA interfaces. The following JTL predicate clearly expresses the required concept:

```
undo_applicable := class {
   setText(String);
   String getText();
   no undo();
};
```

In summary, we propose that in introducing advanced support of genericity and concepts to JAVA, one shall use the JTL syntax as the underlying language for defining concepts. In addition to the two benefits listed above (simple semantics and evaluation, useful definitions not possible in standard JAVA), using JTL also puts intriguing questions of type theory in the familiar domain of logic, since, as mentioned earlier, JTL is based on FOPL*. For example, the question of one concept being contained in another can be thought of as logical implication. Using text book results [15], one can better understand the tradeoff between language expressiveness and computability or decidability. We are currently working on defining a JTL sub-language, restricting the use of quantifiers, which assures decidability of concept containment.

[8] Thus, concepts may be regarded as the generic-programming equivalence of the *Design by Contract* [58] philosophy

5.4 Micro Patterns

A μ-pattern is just like a design pattern, except that it is mechanically recognizable. Previous work on the topic [33] presented a catalog of 27 such patterns; the empirical work (showing that these patterns matched about 3 out of 4 classes) was carried out with a custom recognizer. To evaluate JTL, we used it to re-code each of the patterns.

Fig. 5.5 shows the JTL encoding of the Trait pattern (somewhat similar to the now emerging traits OO construct [69]). In a nutshell, a trait is a base class which provides pre-made behavior to its heirs, but has no state.

The code in Fig. 5.5 should make the details of the pattern obvious: A trait is an abstract class with no instance fields, at least one abstract method, and at least one public concrete instance method which was not inherited from `Object`.

Figure 5.5 The Trait μ-pattern

```
trait := abstract {
    no instance field;
    abstract method;
    public concrete instance method
            !declared_in[Object];
}
```

A programming language researcher could type in the pattern of Fig. 5.5 to quickly find out how many classes in a certain program base are candidates to be implemented as traits.

This example also demonstrates how queries simplify the code. The equivalent PROLOG predicate would have required three recursive calls, probably with the use of auxiliary predicates to implement the three quantifiers in the query.

All patterns were similarly implemented; the specification was never longer than 10 lines. In the course of doing so, we were able to quickly detect ambiguities in the initial textual definition, and check the correctness of the ad-hoc recognizers.

5.5 LINT-like tests.

JTL can be used to express, and hence detect, many undesired programming constructs and habits. On the one hand, JTL's limitation with regard to the inspection of method bodies implies that it cannot detect everything that existing tools [29, 48, 68] can. In its current state, as discussed in Sec. 3.1, JTL cannot detect constructs such as

```
if (C) return true else return false;
```

nor can it easily express numeric limitations (e.g., detecting classes with more than k methods for some constant k).

Yet on the other hand, JTL's expressiveness makes it easy for developers and project managers to improvise and quickly define new rules that are both enforceable and highly self-documenting.

To test this prospect, we a collection of JTL patterns that implement the entire set of warnings issued by Eclipse and PMD (a popular open source LINT tool for JAVA). The only exceptions were those warnings that directly rely on the program source code (e.g., unused **`import`** statements), as these violations are not represented in the binary class file, that we used.

For example, consider the PMD rule LOOSECOUPLING. It detects cases where the concrete collection types (e.g., `ArrayList` or `Vector`) are used instead of the abstract interfaces (such as `List`) for declaring fields, method parameters, or method return values—in violation of the library designers' recommendations. This rule is expressed as a 45-lines JAVA class, and includes a hard-coded (yet partial) list of the implementation classes. PMD does make a heroic effort, but it will mistakenly report (e.g.) fields of type `Vector` for some alien class `Vector` which is not a collection, and was declared outside of the `java.util` package. The JTL equivalent is:

```
loose_coupling := (class|interface) {
  T method | T field | method(*, T, *);
}, T implements /java.util.Collection;
```

It is shorter, more precise, and will detect improper uses of any class that implements any standard collection interface, without providing an explicit list.

5.6 Additional Applications

Several other potential uses for JTL include encapsulation policies and confined types, among others. *Encapsulation policies* were suggested by Scharli et al. [70] as a software construct for defining which services are available to which program modules. Using JTL, both the selection of services (methods) and the selection of modules (classes) can be more easily expressed.

Confined types [14] are another example in which JTL could be used, provided of course that confinement is represented in a form of annotation. We have not yet investigated the question of checking the imperative restrictions of confined types with JTL.

6. Performance

The JTL implementation is an ongoing project involving a team of several programmers. The main challenge is in providing robust and efficient execution environment that can be easily integrated into JAVA tools.

The current implementation, which is publicly available at `http://www.cs.technion.ac.il/jtl`, is an interpreter supporting the language core, including top-down evaluation. It does not yet include a complete implementation for some of the more advanced features described earlier, and defers some of the type checking to runtime.

Work is in progress for a JTL compiler which will translate a JTL program into an equivalent DATALOG program. The DATALOG code can then be processed by an industrial-strength DATALOG engine. State of the art DATALOG research (see e.g., BDDBDDB [78] achieves significant performance, which may benefit JTL as well.

The current implementation uses JAVA's standard reflection APIs for inspecting the structure of a class, and the Bytecode Engineering Library (BCEL) for checking the imperative instructions found within methods. The code spans some 150 JAVA classes that make up a JTL parser, an interpreter, and the implementation of the native predicates that are the basis for JTL's standard library.

On top of this infrastructure there is a text based interactive environment that allows the user to query a given jar file, and an Eclipse plugin that significantly enhances Eclipse's standard search capabilities. Naturally, other application can be easily developed by using JTL's programmatic interface (API).

We will now turn to the evaluation of the performance of this implementation. Our test machine had a single 3GHz Pentium 4 processor with 3GB of RAM, running Windows XP. All JAVA programs were compiled and run by Sun's compiler and JVM, version 1.5.0_06.

In the first set of measurements, we compared the time needed for completing the evaluation of two distinct JTL queries, `q1` and `q2`, defined in Fig. 6.1. Each of the two queries was executed over six increasingly larger inputs, formed by selecting at random 1,000, 4,000, 6,000, 8,000, 10,000 and 12,000 classes from the JAVA standard library, version 1.5.0_06, by Sun Microsystems.

The running time of `q1` and `q2` on the various inputs are shown on Fig. 6.2.

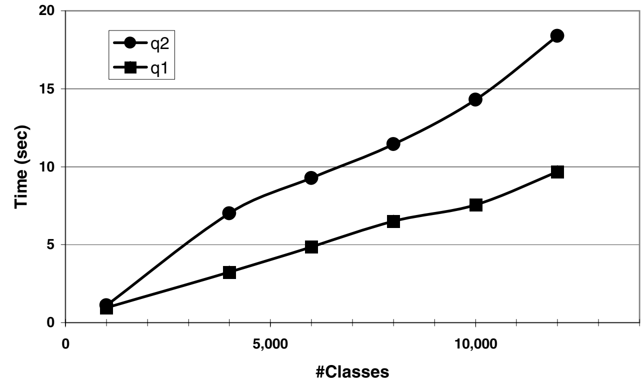
Examining the figure, we see that execution time, for the given programs, is linear in the size of the input. The figure may also suggest that runtime is linear in program size, but this conclusion cannot be true in general, since there are programs of constant size whose output is polynomially large in the input size.

Figure 6.1 JTL queries `q1` and q2. `C.q1` holds if `C` declares a public static method whose return type is `C`; `C.q2` holds if one of the super-classes of `C` is abstract and, in addition, `C` declares a `toString()` method and an **`equals`**`()` method.

```
q1 := eq[T] declares: { public static T (*) ; };

q2 := extends+ : { abstract ; }
   declares : {
      public String toString() ;
      public boolean equals(Object) ;
   } ;
```

Figure 6.2 Execution time of a JTL program vs. input size.



The absolute times are also quite reasonable. For example, it took just about 10 seconds to complete the evaluation of program `q1` on an input of 12,000 classes. Overall, the average execution rate for program `q1` was 1,250 classes per second.

In the second set of measurement we compared JTL's Eclipse plugin with that of JQuery. In a similar manner to JTL, JQuery also tries to harness the power of declarative, logical programming to the task of searching in programs, but (unlike JTL) JQuery expressions are written in a PROLOG-like notation.

Another difference between these two systems relates to the evaluation scheme: JQuery uses a bottom-up algorithm for the evaluation of predicates. As explained in Sec. 4, a bottom-up approach is far from being optimal since it needlessly computes tuples and relations even if they cannot be reached from the given input.

Specifically, JQuery initialization stage, where it extracts facts the from all classes of the program took more than four minutes on a moderate size project (775 classes), which is two orders of a magnitude slower than JTL's initialization phase. Also the first invocation of an individual JQuery query is roughly ten times slower than the corresponding time in JTL.

Therefore, in order to make the comparison fair to JQuery, we broke a user's interaction with the querying system into a sequence of six distinct stages (defined in Fig. 6.3) and compared the performance of JQuery vs. JTL on a stage-by-stage basis.

When running the JTL sessions we used the query `q1` defined earlier. In the JQuery sessions we used query `q1'` (from Fig. 6.4) which is the JQuery equivalent of `q1`.

We timed the JTL and the JQuery sessions on the Eclipse projects representing the source of two open-source programs: JFreeChart[9] (775 classes) and Piccolo[10] (504 classes). The sizes of these projects, in number of classes, are 775 and 504 (respectively).

[9] http://www.jfree.org/jfreechart

[10] http://www.cs.umd.edu/hcil/jazz

Figure 6.3 The sequence of stages used for benchmarking.

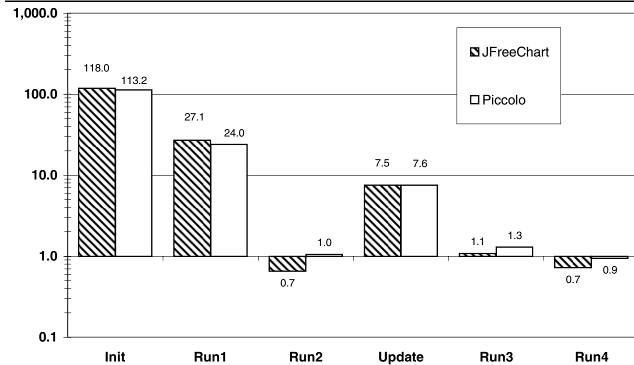
- *Init.* One-time initialization
- *Run1.* First execution of the query
- *Run2.* Second execution of the query.
- *Update.* Updating of the internal data-structure following a slight modification of the source files.
- *Run3.* Third execution of the query.
- *Run4.* Fourth execution of the query.

Figure 6.4 The JQuery equivalent of query q1. Holds for classes C that declare a public static method whose return type is C.

```
        method(?C,?M), returns(?M,?C),
q1' ≜   modifier(?M,static),
        modifier(?M,public)
```

The speedup ratio of JTL over JQuery is presented in Fig. 6.5. The figure shows that JTL is faster in the *Init*, *Run1* and *Update*

Figure 6.5 Speedup of JTL over JQuery, shown on a logarithmic scale. Each pair of columns represents one of the stages defined in Fig. 6.3. Speedup was calculated by dividing the time needed for a stage in the JQuery session with the corresponding time measured from the JTL session.



stages. JTL is about 100 times faster than JQuery at the *Init* stage, and about 25 times faster at the *Run1* stage. JTL was just slightly faster in *Run3*, while JQuery was slightly faster in the *Run2* and *Run4* stages.

As for space efficiency, we predict that a bottom-up evaluator will be less efficient, compared to a top-down evaluator. In particular, we note that running JQuery searches on subject programs larger than 3,000 classes exhausted the memory of the benchmark machine. JTL, on the other hand, was able to process a 12,000-classes project.

7. Discussion and Related Work

Tools and research artifacts which rely on the analysis of program source code are abundant in the software world, including metrics [20] tools, reverse-engineering [10], smart CASE enhancements [47], configuration management [11], architecture discovery [37], requirement tracing [38], AOP) [53], software porting and migration [54], program annotation [3], and many more.

The very task of code analysis per se is often peripheral to such products. It is therefore no wonder that many of these gravitate toward the classical and well-established techniques of formal language theory, parsing and compilation [2]. In particular, software is recurringly represented in these tools in an AST.

JTL is different in that it relies of a flat relational model, which, as demonstrated in Sec. 7.2, can also represent an AST. (Curiously, there were recently two works [36, 56] in which relational queries were used in OO software engineering; however, these pertained to program execution trace, rather than to its static structure.)

JTL aspires to be a universal tool for tool writers, with applications such as specification of pointcuts in AOP, the expression of type constraints for generic type parameters, mixin parameters, selection of program elements for refactoring, patterns discovery, and more.

The community has already identified the need for a general-purpose tool or language for processing software. The literature describes a number of such products, ranging from dedicated languages embedded into larger systems to attempts to harness existing languages (such as SQL or XQUERY [13]) to this purpose. Yet, despite the vast amount of research invested in this area, no single industry standard has emerged.

A well-known example is REFINE [67], part of the *Software Refinery Toolset* by Reasoning Systems. With versions for C, FORTRAN , COBOL and ADA [75], Software Refinery generated an AST from source code and stored them in a database for later searches. The AST was then queried and transformed using the REFINE language, which included syntax-directed pattern matching and compiled into COMMON LISP, with pre- and post-conditions for code transformations. This meta-development tool was used to generate development tools such as compilers, IDEs, tools for detecting violations of coding standards, and more.

Earlier efforts include *Gandalf* [43], which generated a development environment based on language specifications provided by the developers. The generated systems were extended using the ARL language, which was tree-oriented for easing AST manipulations. Other systems that generated database information from programs and allowed user-developed tools to query this data included the *C Information Abstractor* [19], where queries were expressed in the INFOVIEW language, and its younger sibling *C++ Information Abstractor* [40], which used the DATASHARE language.

A common theme of all of these, and numerous others (including systems such as *GENOA* [26], *TAWK* [41], *Ponder* [9], *AST-Log* [24], *SCRUPLE* [66] and more) is the AST-centered approach. In fact, AST-based tools became so abundant in this field that a recent such product was entitled *YAAB*, for "Yet Another AST Browser" [7]. Another category of products is contains those which rely on a relational model. For example, the *Rigi* [60] reverse engineering tool, which translates a program into a stream of triplets, where each triplet associates two program entities with some relation.

Sec. 7.1 compares JTL syntax with other similar products. Sec. 7.2 then says a few words on the comparison of relational- rather than an AST- model, for the task of queering OO languages.

7.1 Using Existing Query Languages

> "*Reading a poem in translation is like kissing your lover through a handkerchief.*"
> H. N. BIALIK (1917)

Many tools use existing languages for making queries. YAAB, for example, uses the Object Constraint Language, OCL, by Rational Software, to express queries on the AST; the *Software Life Cycle Support Environment* (SLCSE) [72] is an environment-generating tool where queries are written in SQL; Rigi's triples representation is intended to be further translated into a relational format, which can be queried with languages such as SQL and PROLOG; etc.

BDDBDDB [78] is similar to JTL in that it uses DATALOG for analyzing software. It is different from JTL in that it concentrates on the specific objective of code optimization, e.g., escape analysis,

and does not further abstract the underlying language. However, the scope of the two for making optimization related analysis.

A more modern system is XIRC [27], where program meta-data is stored in an XML format, and queries are expressed in XQUERY. The JAVA standard reflection package (as well as other bytecode analyzers, such as BCEL) generate JAVA data structures which can be manipulated directly by the language. JQuery [50] underlying is a PROLOG-based extension of Eclipse that allows the user to make queries.

Finally, ALPHA [64] promotes the use of PROLOG queries for expressing pointcuts in AOP. We next compare queries made with some of these languages with the JTL equivalent.

Fig. 7.1(a) depicts an example (due to the designers of XIRC) of using XQUERY to find Enterprise JavaBeans (EJB) which implement `finalize()`, in violation of the EJB specification.

Figure 7.1 Eichberg et. al [27] example: search for EJBs that implement `finalize` in XIRC (a) and JTL (b).

```
subtypes(/class[
      @name="javax.ejb.EnterpriseBean"])
  /method[
      @name = "finalize"
      and .//returns/@type = "void"
      and not(.//parameter)
  ]
```

(a) XIRC implementation of the query (from [27]).

```
class implements /javax.ejb.EnterpriseBean {
   public void finalize();
};
```

(b) The JTL equivalent of (a).

In inspecting the figure, we find that in order to use this language the programmer must be intimately familiar not only with the XQUERY language, but also with the details of the XIRC encoding, e.g., the names of attributes where entity names, return type, and parameters are stored. A tool developer may be expected to do this, probably after climbing a steep learning curve, but its seems infeasible to demand that an IDE user will interactively type a query of this sort to search for similar bugs.

The JTL equivalent (Fig. 7.1(b)) is a bit shorter, and perhaps less foreign to the JAVA programmer.

Fig. 7.1 demonstrates what we call *the abstraction gap*, which occurs when the syntax of the queries is foreign to the queried items.

We next compare JTL syntax with that of JQuery [50], which also relies on Logic programming for making source code queries. Tab. 2 compares the queries used in JQuery case study (extraction of the user interface of a chess program) with their JTL counterparts. The table shows that JTL queries are a bit shorter and resemble the code better.

The JTL pattern in the last row in is explained by the following: To find a method in which one of the type of parameters contains a certain word, we do a pattern match on its argument list, allowing any number of arguments before and after the argument we seek. The desired argument type itself is a regular expression.

The ASPECTJ sub-language for pointcut definition, just as the sub-language used in JAM [5] for setting the requirements for the base class of a mixin, exhibit minimal abstraction gap. The challenge that JTL tries to meet is to do achieve this objective with a more general language.

Fig. 7.2 is an example of using JAVA's reflection APIs to implement a query—here, finding all **`public final`** methods (in a given class) that return an **`int`**.
When compared with Fig. 7.1, we can observe three things:

Figure 7.2 Eliciting public final int methods with the reflection library.

```
public Method[] pufim_reflection(Class c) {
  Vector<Method> v = new Vector<Method>();
  for (Method m : c.getMethods()) {
    int mod = m.getModifiers();
    if (m.getReturnType() == Integer.Type
          && Modifiers.isPublic(mod)
            && Modifiers.isFinal(mod))
        v.add(m);
  }
  return v.toArray(new Method[0]);
}
```

- Fig. 7.2 uses JAVA's familiar syntax, but this comes at the cost of replacing the declarative syntax in Fig. 7.1 with explicit control flow.
- Despite the use of plain JAVA, Fig. 7.2 manifests an abstraction gap, by which the pattern of matching an entity is very different from the entity itself.
- The code still assumes familiarity with an API; it is unreasonable to expect an interactive user to type in such code.

Again, the JTL equivalent, **`public final int`**`(*)`, is concise, avoids complicated control flow, and minimizes the abstraction gap.

We should also note that the *fragility* of a query language is in direct proportion to the extent by which it exposes the structure of the underlying representation. Changes to the queried language (i.e., JAVA in our examples), or deepening the information extracted from it, might dictate a change to the representation, and consequently to existing client code. By relying on many JAVA keywords as part of its syntax, the fragility of JTL is minimal.

There are, however, certain limits to the similarity, the most striking one being the fact that in JTL, an absence of a keyword means that its value is unspecified, whereas in JAVA*, the absence of e.g.,* **`static`** *means that this attribute is off. This is expressed as* `!`**`static`** *in JTL.*

Another interesting comparison with JTL is given by considering ALPHA and Gybels and Brichau's [42] "crosscut" language, since both these languages rely on the logic paradigm. Both languages were designed solely for making pointcut definitions (Gybels and Brichau's work, just as ours, assumes a static model, while ALPHA allows definitions based on execution history). It is no wonder that both are more expressive in this than the reference ASPECTJ implementation.

Unfortunately, in doing so, both languages *broaden* rather than narrow the abstraction gap of ASPECTJ. This is a result of the strict adherence to the PROLOG syntax, which is very different than that of JAVA. Second, both languages make heavy use of recursive calls, potentially with "cuts", to implement set operations. Third, both languages are fragile in the sense described above

We argue that even though JTL is not specific to the AO domain, it can do a better job at specifying pointcuts. (Admittedly, dynamic execution information is external to our scope.) Beyond the issues just mentioned, by using the fixed point-model of computation rather than backtracking, JTL solves some of the open issues related to the integration of the logic paradigm with OO that Gybels, Brichau, and Wuyts mention [16, Sec. 5.2]: The JTL API supports multiple results and there is no backtracking to deal with.

7.2 AST vs. Relational Model

We believe that the terse expression and the small abstraction gap offered by JTL is due to three factors: (i) the logic programming paradigm, notorious for its brevity, (ii) the effort taken in making the logic programming syntax even more readable in JTL, and (iii) the selection of a relational rather than a tree data model.

Task	JQuery	JTL
Finding class "BoardManager"	`class(?C,name,BoardManager)`	**`class`** `BoardManager`
Finding all "main" methods	`method(?M,name,main)` `method(?M,modifier,[public,static])`	**`public static`** `main(*)`
Finding all methods taking a parameter whose type contains the string "`image`"	`method(?M,paramType,?PT)` `method(?PT,/image/)`	`method(*,/?*image?*/,*)`

Table 2. Rewriting JQuery examples [50] in JTL

We now try to explain better the third factor. Examining the list of tools enumerated early in this section we see that many of these rely on the *abstract syntax tree* metaphor. The reason that ASTs are so popular is that they follow the BNF form used to define languages in which software is written. ASTs proved useful for tasks such as compilation, translation and optimization; they are also attractive for discovering the architecture of structured programs, which are in essence ordered trees.

We next offer several points of comparison between an AST based representation and the set-based, relational approach represented by JTL and other such tools. Note that as demonstrated in Sec. 7.2, and as Crew's ASTLog language [24] clearly shows, logic programming does not stand in contradiction with a tree representation.)

1. *Unordered Set Support.* In traditional programming paradigms, the central kind of modules were procedures, which are sequential in nature. In contrast, in JAVA (and other OO languages) a recurring metaphor is the unordered *set*, rather than the *sequence*: A program has a set of packages, and there is no specific ordering in these. Similarly, a package has a set of classes, a class is characterized by a set of attributes and has a set of members, each member in turn has a set of attributes, a method may throw a set of exceptions, etc. Although sets can be supported by a tree structure, i.e., the set of nodes of a certain kind, some programming work is required for set manipulation which is a bit more *natural and intrinsic* to relational structures.

On the other hand, the list of method arguments is sequential. Although possible with a relational model, ordered lists are not as simple. This is why JTL augments its relational model with built-ins for dealing with lists, as can be seen in e.g., the last row of Tab. 2).

2. *Recursive Structure.* One of the primary advantages of an AST is its support for the recursive structures so typical of structured programming, as manifested e.g., in Nassi-Shneiderman diagrams [62], or simple expression trees.

Similar recursion of program information is less common in modern languages. JAVA does support class nesting (which are represented using the `inners` predicate of JTL) and methods may (but rarely do) include a definition of nested class. Also, a class cannot contain packages, etc.

3. *Representation Granularity.* Even though recursively defined expressions and control statements still make the bodies of OO methods, they are abstracted away by our model.

JTL has native predicates for extracting the parameters of a method, its local variables, and the external variables and methods which it may access, and as shown, even support for dataflow analysis. In contrast, ASTs make it easier to examine the control structure. Also, with suitable AST representation, a LINT-like tool can provide warnings that JTL cannot, e.g., a non-traditional ordering of method modifiers.

It should be said that the importance of analyzing method bodies in OO software is not so great, particularly, since OO methods tend to be small [20], and in contrast with the procedural approach, their structure does not reveal much about software architecture [37]. Also, in the OO world, tools are not so concerned with the algorithmic structure, and architecture is considered to be a graph rather than a tree [47].

4. *Theory of Searches.* Relational algebra, SQL, and DATALOG are only part of the host of familiar database searching theories. In contrast, searches in an AST require the not-so-trivial VISITOR design pattern, or frameworks of factories and delegation objects (as in the Polyglot [63] project). This complexity is accentuated in languages without *multi-methods* or *open classes* [18] but occur even in more elaborate languages. Moreover, some questions of attribute grammars (which are essentially what generates AST) are very difficult, e.g., *EXPTIME*-complete [79].

5. *Data Model Complexity.* An AST is characterized by a variety of kinds of nodes, corresponding to the variety of syntactical elements that a modern programming language offers. A considerable mental effort must be dedicated for understanding the recursive relationships between the different nodes, e.g., which nodes might be found as children or descendants of a given node, what are the possible parent types, etc.

The underlying complexity of the AST prevents a placement of a straightforward interface at the disposal of the user, be it a programmatic interface (API), a text query interface or other. For example, in the *Hammurapi*[11] system, the rule "*Avoid hiding inherited instance fields*" is implemented by more than 30 lines of JAVA code, including two **`while`** loops and several **`if`** clauses. The corresponding JTL pattern is so short it can be written in one line:

```
class { field overrides[_] }
```

The terse expression is achieved by the uniformity of the relational structure, and the fact that looping constructs are implicit in JTL queries.

The JTL code in this example is explained as follows: The outer curly parenthesis implicitly loop over all class members, finding all fields among these. The inner ones implicitly loop over all members that this field "overrides". A match (i.e., a rule violation) is found if the inner loop is not empty, i.e., there **`exists`** *one element in the set for which the boolean condition* **`true`** *holds.*

6. *Representation Flexibility.* A statically typed approach (as in Jamoos [34]) can support the reasoning required for tasks such as iteration, lookup and modification of an AST. Such an approach yields a large and complex collection of types of tree nodes. Conversely, in a weakly-typed approach (as in REFINE), the complexity of these issues is manifested directly in the code.

Either way, changes in the requirements of the analysis, when reflected in changes to the kind of information that an AST stores, often require re-implementation of existing code, multiplying the complex reasoning toll. This predicament is intrinsic to the AST structure, since the search algorithm must be prepared to deal with all possible kinds of tree nodes, with a potentially different behavior in different such nodes. Therefore, the introduction of a new kind of node has the potential of affecting all existing code.

[11] http://www.hammurapi.org

In contrast, a relational model is typically widened by adding new relations, without adding to the basic set of simple types. Such changes are not likely to break, or even affect most existing queries.

7. *Caching and Query Optimization.* There is a huge body of solid work on query optimization for relational structures; the research on optimizing tree queries, e.g., XPATH queries, has only begun in recent years. Also, in a tree structure, it is tempting to store summarizing, cached information at internal nodes—a practice which complicates the implementation. In comparison, the well established notion of *views* in database theory saves the manual and confusing work of caching.

8. Conclusions and Further Research

JTL is a novel, DATALOG-based query language designed for querying JAVA programs in binary format. The JTL system can be extended to query programs written in other programming-languages (C#, SMALLTALK [35]), possibly in a different input formats. Such extensions require mostly a rewrite the standard library of native predicates to be replaced with new native predicates which are made to inspect the input at-hand.

We note that the detection of scratch values relies on JAVA's verification process which guarantees certain properties, of the dataflow graph, in every legal method. Therefore, the use of scratch-related predicates over a languages that has a weaker verification process, such as C#, is limited.

Even though termination is always guaranteed (on a finite database) as long as negation is stratified, it is a basic property of FOPL* that other questions are undecidable. For example, it follows from Gödel's incompleteness theorem that it is impossible *in general* to determine e.g., if two queries are equivalent, a query is always empty, the results of one query is contained in another, etc. These limitations are not a major hurdle for most JTL applications. Moreover, there are textbook results [15] stating that such questions are decidable, with concrete algorithms, if the use of quantifiers is restricted, as could be done for certain applications. Still, we believe there is an interesting research challenge in stretching the limitations on the use of negation. One reason for trying to do so is that several classical dataflow analysis problems exhibit non-stratified negation semantics, if expressed in JTL.

JTL sports, whenever possible, a top-down evaluation strategy. Therefore, the amount of information "seen" by JTL's runtime-system during a predicate evaluation is related to the output size *and not* by the size of the full database. Indeed, in many practical situations, the size of the input is significantly smaller than the size of the domain.

For example, when a programmer looks for a class in his source directories, the input is the set of classes found in these directories, where the database may consist of thousands or more classes, including e.g., JAVA's standard runtime library. Thanks to top-down evaluation, the programmer is not penalized for the size of the libraries his program uses.

Moreover, the JTL processor includes a query analyzer, (whose description is beyond the scope of this paper) which determines if a given query is "open" or "close". Intuitively, a query is closed if in computing it, JTL does not need to inspect classes beyond what a JAVA compiler would do. In other words, in the course of processing of a given query, JTL needs only inspect the classes explicitly mentioned in this query or passed to its as parameters, and the classes that they (transitively) depend on these. For example, the query `bad_serialization`, defined in Fig. 8.1, is close, since it can be evaluated while inspecting its input, specifically, on the hidden parameter and the literal `/java.io.Serializable`.

Figure 8.1 A close JTL query

```
bad_serialization :=
   implements /java.io.Serializable {
      no static final long
         field 'SerialVersionUID;
   }
```

The query `classical_interface`, defined in Fig. 8.2, checking whether *all* implementations of the implicit parameter are **`final`**, is open.

Figure 8.2 An open JTL query

```
implemented_by[X] := X implements #;

classical_interface :=
   interface implemented_by: {
      all final;
   };
```

Processing open queries is time consuming. Worse, the output of these queries is non-deterministic, in the sense that it depends on the extent of the software repository available to the processor. This is the reason that the JTL processor warns the programmer against making open queries. As it turns out, JTL queries tend to be close.

We showed that JTL performance exceeds that of a comparable tool. It would still be interested to stretch further JTL scalability and evaluate its performance of close queries of very large programs, typical to the open-source community, or open queries of large libraries and commercial frameworks.

Another scalability concern is query complexity: The examples we provided preserver the spirit of QBE, and are similar to JAVA code. Would this property maintain for increasingly complex queries? We believe that the practice of using small auxiliary queries, such as `implemented_by` of Fig. 8.2, should contribute the cause of preserving the brevity and JAVA-like look of even more complicated queries.

The work on JTL can be continued in the following directions: First, there is the recalcitrant issue of extending JTL to support modifications to the software base. The difficulty here lies with the fact that such changes are expected to preserve the underlying language semantics; in other words, there are complex invariants to which the database under change must adhere. The current "baggage" mechanism is limited precisely for that reason: we are still seeking the balance between sufficient expressive power for string processing and automatic checking that the produced code is correct. The problem becomes even more difficult in dealing with executional code.

Despite these issues, we argue that that JTL can be used, as is, for specifying pre- and post-conditions for existing program transformation systems.

Second, we would like to see a type-safe version of embedded JTL, similar to the work on issuing type safe-embedded SQL calls from JAVA [23, 57] and C# LINQ project[12]. The grand challenge is in a seamless integration, a *linguistic symbiosis* [16] of JTL with JAVA, perhaps in a manner similar to by which XML was integrated into the language by Harden *et al.* [45].

Third, it would be interesting to see if the JTL could be enhanced to examine not only the dataflow of methods, but also their control flow. Even more challenging is the combination of the two perspectives.

Fourth, it might be useful to extend JTL to make queries on the program trace, similarly to PQL [56] or PTQL [36]. This extension

[12] http://msdn.microsoft.com/vcsharp/future/

could perhaps be used for pointcut definitions based on execution stack.

Finally, there is an interesting challenge of finding a generic tool for making language type extensions, for implementing e.g., non-null types [28], read-only types [12], and alias annotations [4]. This could be carried out in the manner described in [32], where the type constraints are specified locally, with two closure conditions: first, a recursively defined constraint on all invocable entities, and second, a condition on allowed modification by inheritance.

The difficulty here lies in the fact that the dataflow analysis we presented is a bit remote from the code. Perhaps the grand challenge is the combination of the brevity of expression offered by JTL with the pluggable type systems of Andreae, Markstrum, Millstein and Noble [6].

Acknowledgements. Inspiring long discussions with Evelina Zarivach greatly helped JTL take its shape. We are also indebted to her for her meticulous read of early drafts of this paper. Comments and encouragements of James Noble are happily acknowledged. Part of the implementation was carried out by Grigory Fridberg.

References

[1] A. V. Aho, B. W. Kernighan, and P. J. Weinberger. *The AWK programming language*. Addison-Wesley series in Computer Science. Addison-Wesley Publishing Company, Reading, Massachusetts, 1988.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.

[3] J. E. Aldrich and C. Chambers. Ownership domains: Separating aliasing policy from mechanisms. In M. Odersky, editor, *Proc. of the Eighteenth European Conference on Object-Oriented Programming (ECOOP'04)*, volume 3086 of *Lecture Notes in Computer Science*, pages 1–25, Oslo, Norway, June 2004. Springer Verlag.

[4] J. E. Aldrich, V. Kostadinov, and C. Chambers. Alias annotations for program understanding. In *Proc. of the Seventeenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'02)*, pages 311–330, Seattle, Washington, Nov. 4–8 2002. OOPSLA'02, ACM SIGPLAN Notices 37(11).

[5] D. Ancona, G. Lagorio, and E. Zucca. Jam—designing a Java extension with mixins. *ACM Transactions on Programming Languages and Systems*, 25(5):641–712, 2003.

[6] C. Andreae, J. Noble, S. Markstrum, and T. Millstein. A framework for implementing pluggable type systems. In *Proceedings of the 21th Annual conference on Object-oriented programming, systems, languages, and applications (OOPSLA'06)*, Portland, Oregon, October 22–26 2006. ACM SIGPLAN Notices.

[7] G. Antoniol, M. D. Penta, and E. Merlo. YAAB (Yet Another AST Browser): Using OCL to navigate ASTs. In *Proc. of the Eleventh International Workshop on Program Comprehension (IWPC'03)*, pages 13–22, Portland, Oregon, USA, May 10-11 2003.

[8] K. Arnold and J. Gosling. *The Java Programming Language*. The Java Series. Addison-Wesley Publishing Company, Reading, Massachusetts, 1996.

[9] D. C. Atkinson and W. G. Griswold. The design of whole-program analysis tools. In *Proc. of the Eighteenth International Conference on Software Engineering (ICSE'96)*, pages 16–27, Berlin, Germany, March 25-30 1996.

[10] L. A. Barowski and J. H. Cross II. Extraction and use of class dependency information for Java. In *Proc. of the Ninth Working Conference on Reverse Engineering (WCRE'02)*, pages 309–318, Richmond, Virginia, USA, Oct. 2002. IEEE Computer Society Press.

[11] L. Bendix, A. Dattolo, and F. Vitali. Software configuration management in software and hypermedia engineering: A survey. In *Handbook of Software Engineering and Knowledge Engineering*, volume 1, pages 523–548. World Scientific Publishing, 2001.

[12] A. Birka and M. D. Ernst. A practical type system and language for reference immutability. In J. M. Vlissides and D. C. Schmidt, editors, *Proc. of the Nineteenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'04)*, pages 35–49, Vancouver, BC, Canada, Oct. 2004. ACM SIGPLAN Notices 39 (10).

[13] S. Boag, D. Chamberlin, M. F. Ferna'ndez, D. Florescu, J. Robie, and J. Siméon. *XQuery 1.0: An XML Query Language*. W3C, 2005.

[14] B. Bokowski and J. Vitek. Confined types. In *Proc. of the Fourteenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'99)*, pages 82–96, Denver, Colorado, Nov.1–5 1999. OOPSLA'99, ACM SIGPLAN Notices 34 (10).

[15] E. Börger, E. Grädel, and Y. Gurevich. *The Classical Decision Problem*. Perspectives of Mathematical Logic. Springer Verlag, 1997.

[16] J. Brichau, K. Gybels, and R. Wuyts. Towards a linguistic symbiosis of an object-oriented and a logic programming language. In J. Striegnitz, K. Davis, and Y. Smaragdakis, editors, *Proc. of the Workshop on Multiparadigm Programming with Object-Oriented Languages (MPOOL'02) at the European Conference on Object-Oriented Programming*, June 2002.

[17] S. Ceri, G. Gottlob, and L. Tanca. *Logic programming and databases*. Springer Verlag, New York, 1990.

[18] C. Chambers. Object-oriented multi-methods in Cecil. In O. L. Madsen, editor, *Proc. of the Sixth European Conference on Object-Oriented Programming (ECOOP92)*, volume 615 of *Lecture Notes in Computer Science*, pages 33–56, Utrecht,the Netherlands, June29–July3 1992. Springer Verlag.

[19] Y.-F. Chen, M. Nishimoto, and C. Ramamoorthy. The C information abstraction system. *IEEE Trans. Softw. Eng.*, 16(3):325–334, Mar. 1990.

[20] T. Cohen and J. Gil. Self-calibration of metrics of Java methods. In *Proc. of the Thirty Seventh International Conference on Technology of Object-Oriented Languages and Systems (TOOLS'00 Pacific)*, pages 94–106, Sydney, Australia, Nov. 20-23 2000. Prentice-Hall, Englewood Cliffs, New Jersy 07632.

[21] T. Cohen and J. Gil. AspectJ2EE = AOP + J2EE: Towards an aspect based, programmable and extensible middleware framework. In M. Odersky, editor, *Proc. of the Eighteenth European Conference on Object-Oriented Programming (ECOOP'04)*, volume 3086 of *Lecture Notes in Computer Science*, pages 219–243, Oslo, Norway, June 2004. Springer Verlag.

[22] M. Consens, A. Mendelzon, and A. Ryman. Visualizing and querying software structures. In *CASCON'91*, pages 17–35. IBM Press, 1991.

[23] W. R. Cook and S. Rai. Safe query objects: statically typed objects as remotely executable queries. In B. N. Gruia-Catalin Roman, William G. Griswold, editor, *Proc. of the Twenty Seventh International Conference on Software Engineering (ICSE'05)*, pages 97–106, St. Louis, MO, USA, May 15-21 2005. ACM Press, New York, NY, USA.

[24] R. F. Crew. ASTLOG: A language for examining abstract syntax trees. In S. Kamin, editor, *Proc. of the First USENIX Conference Domain Specific Languages (DSL'97)*, pages 229–242, Santa Barbara, Oct. 1997.

[25] P. Deransart, L. Cervoni, and A. Ed-Dbali. *Prolog: The Standard: reference manual*. Springer-Verlag, London, UK, 1996.

[26] P. T. Devanbu. GENOA—a customizable, front-end-retargetable source code analysis framework. *ACM Trans. on Soft. Eng. and Methodology*, 8(2):177–212, 1999.

[27] M. Eichberg, M. Mezini, K. Ostermann, and T. Schäfer. XIRC: A kernel for cross-artifact information engineering in software development environments. In *Proc. of the Eleventh Working Conference on Reverse Engineering (WCRE'04)*, pages 182–191, Delft, Netherlands, Nov. 8-12 2004. IEEE Computer Society Press.

[28] M. Fähndrich and K. R. M. Leino. Declaring and checking non-null types in an object-oriented language. In R. Crocker and G. L. S. Jr., editors, *Proc. of the Eighteenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'03)*, pages 302–312, Anaheim, California, USA, Oct. 2003. ACM SIGPLAN Notices 38 (11).

[29] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proc. of the Conference on Programming Language Design and Implementation (PLDI) (PLDI'02)*, pages 234–245, Berlin, Germany, June 17-21 2002. Compaq Systems Research Center.

[30] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Object Technology Series. Addison-Wesley Publishing Company, Reading, Massachusetts, 2000.

[31] R. Garcia, J. Järvi, A. Lumsdaine, J. Siek, and J. Willcock. A comparative study of language support for generic programming. In R. Crocker and G. L. S. Jr., editors, *Proc. of the Eighteenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'03)*, pages 115–134, Anaheim, California, USA, Oct. 2003. ACM SIGPLAN Notices 38 (11).

[32] J. Gil and Y. Eckel. Statically checkable design level traits. In *Proc. of the Thirteenth IEEE Conference on Automated Software Engineering (ASE'98)*, page 217, Honolulu, Hawaii, USA, Nov. 1998. IEEE Computer.

[33] J. Gil and I. Maman. Micro patterns in Java code. In *Proc. of the Twentieth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'05)*, San Diego, California, Oct.16-20 2005. ACM SIGPLAN Notices.

[34] J. Gil and Y. Tsoglin. JAMOOS—a domain-specific language for language processing. *J. Comp. and Inf. Tech.*, 9(4):305–321, 2001.

[35] A. Goldberg. *Smalltalk-80: The Interactive Programming Environment*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1984.

[36] S. Goldsmith, R. O'Callahan, and A. Aiken. Relational queries over program traces. In *Proc. of the Twentieth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'05)*, pages 385–402, San Diego, California, Oct.16-20 2005. ACM SIGPLAN Notices.

[37] I. Gorton and L. Zhu. Tool support for *Just-in-Time* architecture reconstruction and evaluation: An experience report. In B. N. Gruia-Catalin Roman, William G. Griswold, editor, *Proc. of the Twenty Seventh International Conference on Software Engineering (ICSE'05)*, pages 514–523, St. Louis, MO, USA, May 15-21 2005. ACM Press, New York, NY, USA.

[38] O. C. Z. Gotel and A. C. W. Finkelstein. An analysis of the requirements traceability problem. In *Proc. of the First International Conference on Requirements Engineering (ICRE'94)*, pages 94–101, Colorado Springs, Colorado, Apr. 1994. IEEE Computer Society Press.

[39] G. Gottlob, E. Grädel, and H. Veith. Linear time Datalog for branching time logic. In *Logic-Based Artificial Intelligence*. Kluwer, 2000.

[40] J. E. Grass and Y. Chen. The C++ information abstractor. In *Proc. of the USENIX C++ Conference*, pages 265–277, San Fransisco, CA, Apr. 1990. AT&T Bell Laboratories, USENIX Association.

[41] W. G. Griswold, D. C. Atkinson, and C. McCurdy. Fast, flexible syntactic pattern matching and processing. In *Proc. of the Fourth Workshop on Program Comprehension (WPC '96)*, pages 144–153, Washington, DC, 1996. IEEE Computer Society Press.

[42] K. Gybels and J. Brichau. Arranging language features for more robust pattern-based crosscuts. In *Proc. of the Second International Conference on Aspect-Oriented Software Development (AOSD'03)*, pages 60–69, Boston, Massachusetts, USA, Mar. 17-21 2003. ACM Press, New York, NY, USA.

[43] A. N. Habermann and D. Notkin. Gandalf: Software development environments. *IEEE Trans. Softw. Eng.*, 12(12):1117–1127, Dec. 1986.

[44] E. Hajiyev, M. Verbaere, and O. de Moor. Codequest: Scalable source code queries with datalog. In D. Thomas, editor, *Proceedings of the 20th European Conference on Object-Oriented Programming (ECOOP'06)*, volume 4067 of *Lecture Notes in Computer Science*, pages 2–27, Berlin, Germany, 2006. Springer.

[45] M. Harren *et al.* XJ: integration of XML processing into Java. In *WWW Alt. '04: Proc. of the Thirteenth International World Wide Web Conference on Alternate track papers & posters*, pages 340–341, New York, NY, USA, 2004. ACM Press, New York, NY, USA.

[46] A. Hejlsberg, S. Wiltamuth, and P. Golde. *The C# Programming Language*. Addison-Wesley Publishing Company, Reading, Massachusetts, second edition, Oct. 2003.

[47] R. Holmes and G. C. Murphy. Using structural context to recommend source code examples. In B. N. Gruia-Catalin Roman, William G. Griswold, editor, *Proc. of the Twenty Seventh International Conference on Software Engineering (ICSE'05)*, pages 117–125, St. Louis, MO, USA, May 15-21 2005. ACM Press, New York, NY, USA.

[48] D. Hovemeyer and W. Pugh. Finding bugs is easy. *ACM SIGPLAN Notices*, 39(12):92–106, 2004.

[49] ISE. *ISE EIFFEL The Language Reference*. ISE, Santa Barbara, CA, 1997.

[50] D. Janzen and K. D. Volder. Navigating and querying code without getting lost. In *Proc. of the Second international conference on Aspect-Oriented Software Development (AOSD'03)*, pages 178–187, New York, NY, USA, 2003. ACM Press.

[51] J. Järvi, J. Willcock, and A. Lumsdaine. Associated types and constraint propagation for mainstream object-oriented generics. In *Proc. of the Twentieth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'05)*, San Diego, California, Oct.16-20 2005. ACM SIGPLAN Notices.

[52] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *Proc. of the Fifteenth European Conference on Object-Oriented Programming (ECOOP'01)*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–355, Budapest, Hungary, June 2001. Springer Verlag.

[53] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *Proc. of the Eleventh European Conference on Object-Oriented Programming (ECOOP'97)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242, Jyväskylä, Finland, June 9-13 1997. Springer Verlag.

[54] K. Kontogiannis, J. Martin, K. Wong, R. Gregory, H. A. Müller, and J. Mylopoulos. Code migration through transformations. In S. A. MacKay and J. H. Johnson, editors, *Proc. of the Conference of the Centre for Advanced Studies on Collaborative research (CASCON'98)*, page 13, Toronto, Ontario, Canada, Nov. 1998. IBM Press.

[55] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley Publishing Company, Reading, Massachusetts, second edition, 1999.

[56] M. Martin, B. Livshits, and M. S. Lam. Finding application errors and security flaws using PQL: a program query language. In *Proc. of the Twentieth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'05)*, pages 365–383, San Diego, California, Oct.16-20 2005. ACM SIGPLAN Notices.

[57] R. A. McClure and I. H. Krüger. SQL DOM: compile time checking of dynamic SQL statements. In B. N. Gruia-Catalin Roman, William G. Griswold, editor, *Proc. of the Twenty Seventh International Conference on Software Engineering (ICSE'05)*, pages 88–96, St. Louis, MO, USA, May 15-21 2005. ACM Press, New York, NY, USA.

[58] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Englewood Cliffs, New Jersy 07632, Englewood Cliffs, New Jersy, second edition, 1997.

[59] M. Mezini and K. Ostermann. Conquering aspects with Caesar. In *Proc. of the Second International Conference on Aspect-Oriented Software Development (AOSD'03)*, pages 90–100, Boston, Massachusetts, USA, Mar. 17-21 2003. ACM Press, New York, NY, USA.

[60] H. A. Müller and K. Klashinsky. Rigi—A system for programming-in-the-large. In *Proc. of the Tenth International Conference on Software Engineering (ICSE'88)*, pages 80–86, Singapore, Apr. 1988. IEEE Computer Society Press.

[61] G. C. Murphy, M. Kersten, M. P. Robillard, and D. Cubranic. The emergent structure of development tasks. In A. P. Black, editor, *Proc. of the Ninetieth European Conference on Object-Oriented Programming (ECOOP'05)*, volume 3086 of *Lecture Notes in Computer Science*, pages 33–48, Glasgow, UK, July 25–29 2005. Springer Verlag.

[62] I. Nassi and B. Shneiderman. Flowchart techniques for structured programming. *ACM SIGPLAN Notices*, 8(8):12–26, Aug. 1973.

[63] N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An extensible compiler framework for Java. In *Proc. of the Twelfth International Conference on Compiler Construction (CC'03)*, pages 138–152, Warsaw, Poland, Apr. 2003. Springer Verlag.

[64] K. Ostermann, M. Mezini, and C. Bockisch. Expressive pointcuts for increased modularity. In A. P. Black, editor, *Proc. of the Ninetieth European Conference on Object-Oriented Programming (ECOOP'05)*, volume 3086 of *Lecture Notes in Computer Science*, pages 214–240, Glasgow, UK, July 25–29 2005. Springer Verlag.

[65] J. K. Ousterhout. Tcl: An embeddable command language. In *Proc. of the Winter 1990 USENIX Conference*, pages 133–146, Wasington, D.C., Jan. 1990.

[66] S. Paul and A. Prakash. Querying source code using an algebraic query language. In H. A. Müller and M. Georges, editors, *Proc. of the Tenth IEEE International Conference on Software Maintenance (ICSM'94)*, pages 127–136, Victoria, BC, Canada, Sept. 1994. IEEE Computer.

[67] Reasoning Systems. *REFINE User's Manual*, 1988.

[68] N. Rutar, C. B. Almazan, and J. S. Foster. A comparison of bug finding tools for Java. In *Proc. of the Fifteenth International Symposium on Software Reliability Engineering (ISSRE 2004)*, pages 245–256, Saint-Malo, Bretagne, France, Nov. 2–5 2004. IEEE Computer Society Press.

[69] N. Schärli, S. Ducasse, O. Nierstrasz, and A. P. Black. Traits: Composable units of behavior. In L. Cardelli, editor, *Proc. of the Seventeenth European Conference on Object-Oriented Programming (ECOOP'03)*, volume 2743 of *Lecture Notes in Computer Science*, pages 248–274, Darmstadt, Germany, July 21–25 2003. Springer Verlag.

[70] N. Schärli, S. Ducasse, O. Nierstrasz, and R. Wuyts. Composable encapsulation policies. In M. Odersky, editor, *Proc. of the Eighteenth European Conference on Object-Oriented Programming (ECOOP'04)*, volume 3086 of *Lecture Notes in Computer Science*, pages 26–50, Oslo, Norway, June 2004. Springer Verlag.

[71] C. Smith and S. Drossopoulou. Chai: Traits for Java-like languages. In A. P. Black, editor, *Proc. of the Ninetieth European Conference on Object-Oriented Programming (ECOOP'05)*, volume 3086 of *Lecture Notes in Computer Science*, Glasgow, Scotland, July 25–29 2005. Springer Verlag.

[72] T. Strelich. The Software Life Cycle Support Environment (SLCSE): a computer based framework for developing soft. sys. In *Proc. of the Third ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments (SDE'88)*, pages 35–44, Boston, Massachusetts, 1988. ACM Press, New York, NY, USA.

[73] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley Publishing Company, Reading, Massachusetts, third edition, 1997.

[74] B. Stroustrup and G. D. Reis. Concepts—design choices for template argument checking. ISO/IEC JTC1/SC22/WG21 no. 1536, 2003.

[75] S. T. Taft and R. A. Duff, editors. *Ada 95 Reference Manual, Language and Standard Libraries, International Standard ISO/IEC 8652: 1995(E)*, volume 1246 of *LNCS*. Springer Verlag, 1997.

[76] M. Tatsubori, S. Chiba, K. Itano, and M.-O. Killijian. Openjava: A class-based macro system for java. In W. Cazzola, R. J. Stroud, and F. Tisato, editors, *Proc. of the First OOPSLA Workshop on Reflection and Software Engineering*, volume 1826 of *Lecture Notes in Computer Science*, pages 117–133, Denver, CO, USA, Nov. 1999. OOPSLA'99, Springer Verlag.

[77] A. van Gelder, K. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991.

[78] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analyses using binary decision diagrams. In *Proc. of the Conference on Programming Language Design and Implementation (PLDI) (PLDI'04)*, pages 131–144, Washington, DC, June 9-11 2004. ACM Press, New York, NY, USA.

[79] P.-C. Wu. On exponential-time completeness of the circularity problem for attribute grammars. *ACM Transactions on Programming Languages and Systems*, 26(1):186–190, 2004.

[80] M. M. Zloof. Query By Example. In *Proceedings of the National Computer Conference*, pages 431–438, Anaheim, CA, May 1975.