

Guarded Program Transformations using JTL

Tal Cohen^{1*}, Joseph (Yossi) Gil², and Itay Maman²

¹ Google Haifa Engineering Center

² Department of Computer Science, Technion—Israel Institute of Technology
talcohen@google.com, {yogi, imaman}@cs.technion.ac.il

Abstract. There is a growing research interest in employing the logic paradigm for making *queries* on software in general, and OOP software in particular. We describe a side-effect-free technique of using the paradigm for the general task of *program transformation*. Our technique offers a variety of applications, such as implementing generic structures (without erasure) in JAVA, a Lint-like program checker, and more. By allowing the transformation target to be a different language than the source (*program translation*), we show how the language can be employed for tasks like the generation of database schemas or XML DTDs that match JAVA classes.

The technique is an extension of JTL (Java Tools Language, [12]), which is a high-level abstraction over DATALOG. We discuss the JTL-to-DATALOG compilation process, and how the program transformation extension can be added to JTL without deviating from the logic paradigm, and specifically without introducing side-effects to logic programs.

Key words: Declarative Programming, Program Transformations

1 Introduction

The logic paradigm is attracting increasing attention in the software engineering community, e.g., CodeQuest [20] and JQuery [22] and even dating back to the work of Minsky [29] and law governed systems [30]. The aspect-orientation school is also showing a growing interest in this paradigm, with work on aspect-oriented logic meta programming [15], the ALPHA system [31], Carma [7], LOGICAJ [33], the work of Gybels and Kellens [19] and more.

Our own work in this area, JTL (the Java Tools Language) [11–13], is a logic-paradigm language designed for the purpose of making queries over the static structure of JAVA programs. JTL’s design took a special effort to adapt the logic to the task at hand, and to provide a Query-By-Example [38] flavor to queries.

Much (though not all) of this previous art is characterized by use of the logic paradigm for *querying* code, rather than code generation. (Exceptions include TYRUBA and LOGICAJ, which concentrate in code weaving—a kind of transformation.)

This paper describes a technique of using the paradigm for the *production of output*. The technique is demonstrated through a JTL language extension, which hinges on associating each predicate with a “baggage” string variable (or more generally, an array of such variables). This variable can be interpreted as the query’s output, or *result*.

* Work done while the author was at the Technion.

There are clear and simple rules for computing the result of compound predicates from the results of their components. Chief to this extension is the introduction of *tautology predicates*; a tautology is a predicate which always holds for all settings of the arguments, but returns a string result describing the parameters.

Specifically, the JTL primitive- and library-predicates were redesigned so that each predicate returns a string which is the JAVA code fragment that best describes this string. Thus, predicate `final` returns the string "final" and the pattern `final abstract` returns the string "final abstract". Moreover, many tautologies, such as `visibility` (returning either of "public", "protected", "private" or the empty string, depending on the argument's visibility) were added to the library.

Complementing tautologies is a construct that allows the JTL program to *discard* the result of a JTL expression (that is: the result will not appear in the result of the enclosing expression). This allow a JTL predicates to be conceptually divided into two components: the *guard* [16], which describes the pre-requisites for the transformation, and the *transformer*, which is the clockwork behind the production of output for the matched code.

The result of a query evaluation can then be used for many purposes. We describe here the following applications of this extension:

1. *Program Transformation*. Perhaps the most obvious application of the string result is to generate JAVA code. e.g., it is straightforward to write a small JTL program that, given an interface, generates a class boilerplate that implements this interface, including method bodies to be refined by a programmer. Many other refactoring tasks are readily implemented as JTL transformations. Further, one can use JTL to implement mixins [5] in JAVA, or even genericity (which does not suffer from the restrictions due to erasure semantics and is as powerful as MIXGEN and NEXTGEN [2]).
2. *Translation*. The literature [35, 37] distinguishes between *rephrasing* transformations (such as the refactoring steps discussed above) and *translating* transformations. Examples for translation include using JTL to generate SQL code that defines a database scheme corresponding to a given JAVA class. In the same fashion, it is possible to generate an XML datatype definition out of a class structure. Some other translation applications which JTL can handle include a Lint-like tool detecting coding convention violations and potential bugs ("code smells"), and a documentation tool which may elicit a JavaDoc skeleton out of class signatures.
3. *Using JTL as a limited AOP Language*. The relationship between program transformation and aspects is a fascinating subject [15, 17, 18, 25, 26]. The community is aware of this relationship ever since the original introduction of AOP [23], which stated that "*some transformations are aspectual in nature*".

We give examples showing that JTL can be used to produce JAVA code that augments the original code, or even replaces it entirely, and that this code production supports aspect orientation. While in some senses, the code is not as elegant as in "pure" aspect-oriented programming languages, JTL does introduce discipline into program transformations, including the definition of pointcuts to which advice are applied. And because the pointcuts themselves are defined in JTL's powerful

query language, we find that this toy AOP language has some capabilities that are unmatched even by ASPECTJ.²

In a sense, these abilities are not surprising: software that can generate textual output can, in principle, generate programs in any desired programming language. One of the important factors that make our approach better for this task is that the production of output is an *implicit* part of the pattern matching process. Programmer intervention is required only where the defaults are not appropriate.

Contributions. The power of a purely declarative system for querying a codebase is generally acknowledged (e.g., [12, 20]). This paper shows how this power can be extended to the output generation domain. Since any transformation program must provide an orderly selection of transformed elements, there is a great advantage in using the same language for both tasks. Novel aspects and contributions of this work include:

1. *Declarative Non-Intrusive High-Level String Generation.* We demonstrate the power of a *declarative* specification of string output generation, directed by the process of pattern matching. Towards this end we introduce specific high-level mechanisms such as list and set iterations, *tautologies*, multiple output streams, and more. These additions do not break the semantics of existing JTL code.
2. *Simplification of the Output Validation Process.* The process of program transformation is in fact a kind of software composition. It is therefore paramount to offer a way that will validate the output of a program transformation tool ahead of time. As explained in §6, the clean, logic paradigm based output generation should contribute to output validation.
3. *Representing transformations using DATALOG.* We show how JTL's high-level queries can be compiled into pure DATALOG, including the generation of string output. This is achieved without resorting to artifacts such as side-effects, which are alien to logic programming.

We note that this version of JTL does not deal yet with statements which may appear in method bodies. This is not an inherent limitation: We can almost technically generalize our results (as hinted in [12]), and use our underlying relational model to express recursive statement structure and more generally any AST. There is of course an engineering challenge in doing that without blowing up the underlying library.

Outline. The remainder of this paper is organized as follows: We start with a short introduction to JTL (§2), and proceed with a description of the new mechanisms for producing output (§3). §4 demonstrates some applications of the new language capabilities. In §5 we show how these mechanisms affect the JTL-to-DATALOG compilation process. The output validation problem is briefly discussed in §6. Finally, §7 presents related work and concludes.

2 A Brief Overview of JTL

This section overviews the JTL core, at a level sufficient for reading the examples given in this paper. Readers familiar with the language may skip this section; readers look-

² See [12] for a discussion of JTL's pointcut-expression capabilities.

ing for a more detailed description are referred to previous works describing the language [11, 12], and the JTL web-site at www.cs.technion.ac.il/jtl.

Simple predicates. Just like other languages in the logic paradigm, the basic constructs of JTL are *predicates*, also called *patterns*. Many (but not all) JTL primitives are JAVA keywords; each such keyword matches precisely those program elements which are declared by it. *Conjunction* is denoted with a space or a comma, *disjunction* is denoted by a vertical bar, and *negation* by an exclamation mark; operator precedence is the usual, while square brackets may be used to override precedence. For example, the expression `[public | protected] !static int` matches non-*static* class members (both fields and methods) of type `int` that are either `public` or `protected`.

Predicate definitions are used to name expressions. For example,

```
instance := !static;
service := public instance method;
```

names the `instance` and `service` predicates, which can now be used just like primitive predicates in composing expressions.³

Unary predicates (such as those above) operate on a single implicit parameter, the *subject*, which is denoted `#` (similar to the JAVA keyword `this`). JTL also supports binary (and higher-arity) predicates, which accept an explicit *argument* (or arguments) in addition to the implicit subject. For example, the binary primitive predicate `declares[M]` holds if `#` is a class or interface which declares the member `M`. Similarly, `extends[S]` holds if `s` is the superclass of `#`.

As in DATALOG, variables always begin with an upper-case letter, whereas predicate names must begin in lower-case. JTL allows using any number of the special characters `+`, `*` and `'` in the suffix of identifiers; the standard JTL library thus defines `extends+` as the transitive closure of `extends` and `extends*` as the reflexive version of `extends+`. The underscore symbol (“`_`”) represents an unnamed variable; it is useful if we do not care about a certain position in the relation that a predicate matches.

Developers can define their own binary predicates; e.g., given

```
interfaceof[C] := C.class C.implements[#];
```

the term `I.interfaceof[T]` holds if `T` is a class that implements interface `I`. The above can be re-written more concisely using several syntactic aids: the *subject-chaining* operator, “`&`”; dropping the square brackets, which are optional in binary predicates; and omitting the optional dot. The result is: `interfaceof C := C class & implements #;`

This allows for readable, English-like predicates. Yet because the arity of predicates is fixed (even for overloaded versions), no syntactic ambiguity arises.

JTL employs variable binding similar to that of DATALOG. The `common` predicate in the following example holds if `#` and `T` have a common super-interface:

```
common T := implements X, T implements X;
```

JTL’s Kind System The examples seen so far can be readily rewritten in DATALOG, possibly with the aid of auxiliary predicates. One may accuse JTL for being syntactic sugar for DATALOG. A more fair description is that JTL offers higher level abstractions on top of core DATALOG, and is compiled to DATALOG, just as EIFFEL is compiled to C. One such abstraction is a type system.

³ Predicates `instance` and `service` are in fact part of JTL’s rich standard library.

Types in JTL are called *kinds*. JTL basic kinds include `TYPE` (the kind of JAVA classes and interfaces), `MEMBER` (class members), `STRING` (a simple string value), `PACKAGE` and more. Compound kinds are lists, including `MEMBERS` (a list of `MEMBERS`), `STRINGS`, etc. See §5.3 for a discussion of the implementation of lists in JTL.

Iteration Constructs The loop control structure of imperative programming languages is almost always translated to recursive predicate definitions involving multiple rules in logic programming. JTL's *quantifiers* are a higher level-abstraction, which allows a more concise and clear expression of such situations. Quantifiers are conditions over sets, e.g., a *universal* quantifier checks that all members of a set satisfy a given predicate.

Set Quantifiers. The following predicate, for example, checks that *all* interfaces that the subject class implements are `public`:

```
all_interfaces_public := implements: { all public; };
```

The computation here involves two stages: (a) generating a set, and (b) applying a quantified condition to the entire set. The “:” character that follows the binary predicate `implements` turns this predicate into a *generator*, which returns the set of all `xs` such that the expression `#.implements[x]` holds. Quantifier application is then carried out inside the curly brackets. Specifically, `all public` is a *set condition* which checks that predicate `public` holds for all members of this set.

In a sense, the curly brackets are a looping construct iterating over the elements of the generated set. The current element of the iteration serves as the subject of the quantified condition. The internal curly brackets scope hides the subject variable of the external scope; therefore, the condition `public` from the previous example will be evaluated against each of the values in the generated set.

JTL's quantifiers include: `has` (alias `exists`), `all`, `no`, `one`, `many`, and `empty` (which is equivalent to `all false`). A condition without a quantifier is understood as an existential quantifier. If the curly brackets contain no quantifiers, `empty;` is assumed. If a curly brackets scope has no preceding generator, then the default `members:` is inferred.

Argument Predicates and List Quantifiers. The list of arguments to a method can be examined by an *argument list predicate*. The most simple argument list is the empty list, which matches methods and constructors that accept no arguments. An asterisk is an arguments list predicate which matches a sequence of zero or more types. Thus, the standard-library predicate `invocable := (*);` matches members which may take any number of arguments, i.e., constructors and methods, but not fields.

Another example of list quantification is `(int, *, int)`, which means that the first and last elements are of type `int`, and any number of parameters may occur in between.

Other Predicates *Name predicates*, such as `'finalize'`, are enclosed in single quotes, and are matched against a class or member's name. Standard JAVA regular expressions can be used, except that the wildcard character is denoted by a question mark rather than a dot, so `'set?*` matches all members whose name begins with “set”.

To match the type of a class member, use *type predicates*, which are regular expression enclosed between forward slashes; e.g., predicate `/java.util.?*/ method` matches all methods with a return type from the `java.util` package (or its sub-packages).

The underscore is a special type predicate which matches any type. Hence, the expression `public _ (_, int, *)` matches public methods that accept an `int` as their second argument, and return any type (but not constructors, which have no return type).

Type predicates can also be used as literals of kind `MEMBER`, i.e., as actual parameters to predicates—as in `interface extends /Serializable/`, which matches any interface that extends the standard `Serializable` interface.

Additional signature predicates exist for testing the `throws` list of a method or constructor, and the annotations attached to any language element.

3 JTL Extensions for Program Transformation

In a nutshell, JTL is a simply typed high level language in the logic paradigm, whose underlying model is that of DATALOG augmented with well founded negation, but without function symbols. That is to say that JTL (unlike PROLOG) uses variable *binding*, rather than *unification*, and predicates' parameters, just like other variables, are atomic. Abstraction mechanisms over plain DATALOG offered by JTL include features such as set and list predicates, cascades and scoped definitions. The language is object-oriented in the (limited) sense that every predicate has an implicit receiver. Primitive types in JTL include `STRING`, `MEMBER` (representing e.g., methods and fields) and `TYPE` (representing e.g., `classes` and `interfaces`); there are no compound types.

The observation that enabled the support of transformations is that a predicate in the logic paradigm can include variables to be used for output, their value being set as part of the standard process of evaluating the predicate. This is also true for JTL predicates. Because JTL supports string variables (the `STRING` kind), it is possible to harness such output-only variables for the generation of string output. To do so, all one needs to do is introduce an additional parameter of this kind to every relevant predicate, and appropriate terms in the predicate's body to bind the value of this variable as part of the pattern-matching process. However, the process of managing this “baggage” variable is tiresome and error prone.

This section describes a JTL language extension which automatically manages this baggage information. §4 will give a number of detailed applications of the mechanism, ranging from program transformation tools to an AOP language.

The principle behind the extension is simple: every JTL predicate implicitly carries with it an unbounded array of baggage anonymous `STRING` variables, which are computed by the predicate. These variables are output only—an invocation of a predicate cannot specify an initial value for any of them. The compilation process translates into DATALOG only baggage which is actually used. Thus, plain JTL queries, as well as their DATALOG equivalent, are not changed, since no baggage variables are used.

In most output-producing applications, only the first baggage variable, called the *standard output* or just the *output* of the predicate, is used. The output parameter is sometimes called the “returned value” in the context of program transformation.

The description begins (§3.1) with the assumption that there is indeed only one such baggage; the subsequent §3.2 explains how multiple baggage variables are managed. §3.3 shows how escaping inside string literals can be used for producing more expressive output. The most important feature of baggage processing—the iterative production of output with quantifiers—is the subject of §3.4.

3.1 Simple Baggage Management

The essence of the baggage extension is that the output of a compound predicate is constructed by default from its component predicates. Since the initial purpose of our extension was the production of JAVA code, the library was so designed that the output of JTL predicates that are also JAVA keywords (e.g., `synchronized`) return their own name on a successful match. Other primitive predicates (e.g., `method`) return an empty string. Type and name patterns return the matching type or name. The fundamental principle is that whenever possible, any predicate returns the text of a JAVA code fragment that can be used for specifying the match.

The returned value of the conjunction of two predicates is the concatenation of the components. By default, this concatenation trims white spaces on both ends of the concatenated components, and then injects a single space between these. Disjunction of two predicates returns the string returned by the first predicate that is satisfied. Thus, for example, the predicate `public static [int | long] field 'old?*'` can be applied to some field called `oldValue`, in which case it will generate an output such as `"public static int oldValue"`.

String literals are valid predicates in JTL, except that they always succeed. They return in the output their own value. By using strings, predicates can generate output which is different from echoing their building blocks. For example, the pattern `class '?*' "extends Number"` generates, when applied to class `Complex`, the output `"class Complex extends Number"`. The string literal in the pattern does not present any requirement to the tested program element, and the string result need not be an echo of that element. The pattern above, for example, will successfully match class `String`, which does *not* extend `Number`.

String literals are just one example of what we call *tautologies*: predicates which hold for any value of their parameters. Tautologies are used solely for producing output. The most simple tautology is the predicate `nothing`, which returns the empty string, i.e., `nothing := ""`. With the language extension, the JTL library was extended with many such tautologies, e.g., `visibility` mentioned above in §1, or `multiplicity`, defined as `static | nothing`.

Other tautologies in the library include `modifiers`, returning the string of all modifiers used in the definition of a JAVA code element; `signature`, returning the type, name, and parameters of methods, or just the type and name of fields; `header`, including the modifiers and signature; the (primitive) `torso`, returning the body (without the head and embracing curly brackets) of a method or a class; and `preliminaries`, returning the package declaration of a class, etc. Tautology `declaration`, whose baggage is the full definition of a program element, is useful for the exact replication of the matched element. We have (for classes and methods, but not for fields)

```
declaration := preliminaries header "{" torso "}";
```

The following demonstrates how tautology `header` and several other auxiliary tautologies are defined:

```
header := modifiers declarator '?*' parents; (1)
modifiers := concreteness strictness visibility ...;
concreteness := abstract | final | nothing;
strictness := strictfp | nothing;
...
```

```

declarator := class | interface | enum | @interface;
parents := superclass optional_interfaces;
superclass := extends T ![T is Object] | nothing;
...

```

(The declaration of `optional_interfaces` is shown below, in §3.4.) The actual definitions are a bit more involved, since they have to account for annotations and generic parameters, and must have overloaded versions for elements of kind `MEMBER`.

The negation operator, `!`, discards any output generated by the expression it negates. For example, `!static` will generate an empty string when successfully matched. Thus `multiplicity` can be also defined as `static | !static`.

Finally, if a JTL main query returns several answers, then the output of the whole program is obtained by concatenating the outputs of each result, but the order is unspecified. Thus, if a JTL query matches all methods called by a given method, then the order by which these methods are generated is unspecified, and hence the concatenation of the string results can be in any order. In most cases however, JTL programs are written to produce a single output, or be applied in a setting where only one output makes sense.

3.2 Multiple Baggage

It is sometimes desirable to suppress the output of one or more constituents of a pattern, even if they are not negated. This can be done by prepending the percent character, `%`, to the expression. For example, the predicate

```
%public %static %final _ '?*' (2)
```

will match any public static final element, but print only its type and name, without the modifiers that were tested for. Predicate (2) can also be written using square brackets:

```
%[public static final] _ '?*' (3)
```

The suppression syntax is in fact one facet of a more complex mechanism, which allows predicates to generate multiple string results, directed to different *output streams*. By default, any string output becomes part of string result 1, which is normally mapped to the standard output stream (`stdout` in Unix jargon). Also defined are string result 0, which discards its own content (`/dev/null`), and string result 2, the standard error stream (`stderr`).

To direct an expression's string output to a specific string result, prepend a percent sign and the desired string result's number to the expression. A percent sign with no number, as used above in (2) and (3), defaults to `%0`, i.e., a discarded string result.

For example, consider the following predicate:

```
testClassName := %2[ class '[a-z]?*' "begins with a lowercase letter." ];
```

If matched by a class, it will send output to string result 2, i.e., the standard error stream; possible output can be "class badlyNamed begins with a lowercase letter". If, however, the expression is not matched (in this case, because the class does not begin with a lowercase letter), no output is generated.

By using disjunction, we can present an alternative output for those classes that do not match the expression; for example:

```
testClassName := %2[class '[a-z]?*' "begins with a lowercase letter."
| [class '?*' "is properly named."]; (4)
```

Because it is not directed to any specific stream, the string result of the second part of the predicate is directed to the standard output. As explained below in §5.4, the disjunction operator's output is evaluated in a non-commutative manner, so that its

right-hand operand can generate output only if its left-hand one yielded false. Thus, predicate (4) will generate exactly one of two possible messages, to one of two possible output streams, when applied to a class. The query `testClassName[X]` is a tautology for classes, matching any class `x`; its *output*, however, depends on `x`'s name.

A configuration file binds any string result generated by a JTL program to specific destinations (such as files). Multiple output streams can be used in a single translation job. However, to process a large codebase with multiple classes, there is no need to define an output stream per input file; rather, Ant-like tools can be used to run JTL once per input file.

JTL also includes mechanisms for redirecting the string result generated by a subexpression into a different string result in the calling expression, or even to bind string results to variables. The syntax `%n>m p` will redirect the string result of predicate `p` in output stream `n` into output stream `m` of the caller. For example, the expression

```
[%2>1 p1] | "failed" (5)
```

will yield the string result that `p1` sends to output stream 2, in output stream 1. If `p1` fails, (5) will generate the output `failed`. However, if `p1` succeeds without generating any output in stream 2 (e.g., it generates no output at all, or output to other streams only) then (5) will generate no output. To bind string results to a variable, the syntax `%n>V p` can be used, binding the output of predicate `p` in output stream `n` to variable `V` of the caller. Thus, writing `%2>Error refactor` will assign the output stream 2 of predicate `refactor` to variable `Error`. Redirection into a variable is only permitted if the variable is output-only—that is, if no other assignments into it occur in the product. To determine whether a variable is output-only, we rely on JTL's pre-analysis stage [10] designed to decide whether a query result is bounded (e.g., returning all class ancestors), or unbounded in the sense that it depends of whole program information (e.g., returning all class's descendants). A by-product of this analysis tells decides whether a variable is output only.

If necessary, file redirection can also be achieved from within JTL just as in the AWK programming language. E.g., `%2>"/tmp/err.log" refactor` will redirect the standard error result of `refactor` to the file named `/tmp/err.log`. It is also possible to redirect output into a file whose name is computed at runtime, as explained at the conclusion of the following §3.3.

Admittedly, redirection syntax degrades from the language elegance, but it is expected to be used rather rarely.

3.3 String Literals

Baggage programming often uses string literal tautologies. Escaping in these for special characters is just as within JAVA string literals. For example, `"\n"` can be used to generate a newline character. An easier way to generate multi-line strings, however, is by enclosing them in `\[... \]`, which can span multiple lines.

When output is generated, a padding space is normally added between every pair of strings. However, if a plus sign is added directly in front of (following) a string literal, no padding space will be added before (after) that string. For example, the predicate `class "New"+ '?*'` will generate the output `"class NewList"` when applied to the class `List`.

The character `#` has a special meaning inside JTL strings; it can be used to output the value of variables as part of the string. For example, the predicate

```
class "ChildOf"+ ['?*' is T] "extends #T" (6)
```

will yield “class ChildOfInteger extends Integer” when applied to `Integer`. The first appearance of `T` in predicate (6) captures the name of the current class into the variable; its second appearance, inside the string, outputs its value.

When applied to JAVA types, a name pattern returns (as a string value) the short name of a class, whereas a type pattern returns the fully-qualified class name. We can therefore write (6) as `class "ChildOf"+ ['?*'] "extends" _` to obtain “class ChildOfInteger extends java.lang.Integer”.

The sharp character itself can be generated using a backslash, i.e., “\#”. To output the value of `#` (the current receiver) in a string, just write “#”. For example, the following binary tautology, when applied to an element of kind `MEMBER`, outputs the name of that element with the parameter prepended to it: `prepend[Prefix] := "#Prefix+#";`.

In case of ambiguity, the identifier following `#` can be enclosed in square brackets. More generally, `#` followed by square brackets can be used to access not only variables, but also output of other JTL expressions.⁴ For example, the following tautology returns a renamed declaration of a JAVA method or field:

```
rename[Prefix] := modifiers _ prepend[Prefix] [
  method (*) throwsList "{ #[torso] }"
  | field "= #[torso];"
];
```

3.4 Baggage Management in Quantifiers

In the `rename` predicate example above, the term `(*)` outputs the list of all parameters of a method. Set and list quantifiers generate output like any other compound predicates. Different quantifiers used inside the generated scope generate output differently. In particular, `one` will generate the output of the set or list member that was successfully matched; `many` and `all` will generate the output of every successfully matched member; and `no` generates no output. The extension introduces one additional quantifier, which is a tautology: writing `optional p;` in a quantification context prints the output of `p`, but only if `p` is matched.

For example, the following predicate will generate a list of all fields and methods in a class that were named in violation of the JAVA coding convention:

```
badlyNamedClassMembers := %class %{
  [field|method] '[A-Z]?*' "is badly named."; (7)
  %}
```

By default, the opening and closing characters `()` or `{ }` print themselves; their output can be suppressed (or redirected) by prepending a `%` to each character, as above.

Two (pseudo) quantifiers, `first` and `last`, are in charge of producing output at the beginning or the end of the quantification process. The separator between the output for each matched member (as generated by the different quantifiers) is a newline character in set quantifiers, or a comma in the case of list quantifiers. This can be changed using another pseudo-quantifier, `between`. The tautology `optional_interfaces` used in the above definition of `header` (1) requires precisely this mechanism:

⁴ Note that using JTL expressions inside a string literal may mean that the literal is not a tautology, e.g., “#[public|private]” is not a tautology.

```
optional_interfaces := implements: %{
    first "implements";
    exists _; -- and names of all super interfaces
    between ","; -- separated by a comma
    last nothing; -- and no ending text
%}
| nothing;
```

Since we use the `exists` quantifier, the entire predicate in the curly bracket fails if the class implements no interfaces—in which case the “`first`” string “implements” is not printed; if this is the case, then the `| nothing` at the end of the definition ensures that the predicate remains a tautology, printing nothing if need be.

4 Transformation Examples

This section shows how JTL’s baggage can be used for various tasks of program transformation. The description ignores the details of input and output management; the implicit assumption is that the transformation is governed by a build-configuration tool, which directs the output to a dedicated directory, orchestrates the compilation of the resulting source files, etc. This makes it possible to apply a JTL program in certain cases to replace an existing class, and in others, to add code to an existing software base.

4.1 Using JTL in an IDE and for Refactoring

We have previously described [12] the JTL Eclipse plug-in, and how it can be used to detect programming errors and potential bugs. It should be obvious that baggage output makes it possible for JTL to not only detect such problems, but also provide useful error and warning messages. Pattern (7) in the previous section shows an example.

JTL can also be put to use in refactoring services supplied by the IDE.⁵ The following pattern extracts the public protocol of a given class, generating an `interface` that the class may then implement:

```
elicit_interface := %class -- Guard: applicable to classes only
modifiers "interface" prepend["P."] -- Produce header
{ -- iterate over all members
    optional %public !static method header "," ;
};
```

We see in this example the recurring JTL programming idiom of having a *guard* [16] which checks for the applicability of the transformation rule, and a *transformer* which is a tautology. (Note that by convention, the output of guards is suppressed, using the percent character.) The interface is generated by simply printing the header declaration of all public, non-static methods.

The converse IDE task is also not difficult. Given an interface, the following JTL code shows how, as done in Eclipse, a prototype implementation is generated:

```
defVal := %boolean "false" | %primitive "0" | %void nothing | "null";
gen_class := %interface -- Guard: applicable to interfaces only
modifiers "class" prepend["C."] "implements #" {
    header \[ {
        return #[defVal];
    } \]
};
```

The above also demonstrates how JTL can be used much like output-directed languages such as PHP and ASP: output is defined by a multi-line string literal, into which, at

⁵ We note, however, that some refactoring steps exceed JTL’s expressive power.

selected points, results of evaluation are injected. Here, the value of the tautology `defVal` is used to generate a proper default returned value.

4.2 JTL as a Lightweight AOP Language

With its built-in mechanism for iterating over class members, and generate JAVA source code as output, it is possible to use JTL as a quick-and-dirty AOP language. The following JTL predicate is in fact an “aspect” which generates a new version of its class parameter. This new version is enriched with a simple logging mechanism, attached to all public methods by means of a simple “before” advice.

```

1 loggingAspect := %class header declares: {
2   targetMethod := public !abstract method; -- pointcut definition
3
4   %targetMethod header \[ {
5     System.out.println("Entering method #");
6     #[torso]
7   } \]
8 | declaration;
9 }
```

The local predicate `targetMethod` defines the kinds of methods which may be subjected to aspect application—in other words, it is a guard serving as a pointcut definition. The condition in the existential quantifier is a tautology; therefore, output will be generated for every element in the set. The first branch in the tautology, its guard (line 3), is the term `%targetMethod`.

If the member is matched against the guard, the method’s header is printed, followed by a *modified* version of the method body. If, however, the member does not match the `targetMethod` pointcut, the disjunction alternative `declaration` will be used—i.e., class members that are not concrete public methods will be copied unchanged.

Having seen the basic building blocks used for applying aspects using JTL, we can now try to improve our logging aspect. For example, we can change the logging aspect so that it prints the actual arguments as well, in addition to the invoked method’s name. To do so, we define the following tautology:

```

actualsAsString := % (
  first \["( "+ \]; last \["+" \]; between \["+", "+ \];
  argName; -- at least one; iterate as needed
%)
| "()"; -- no arguments
```

Given a method signature with arguments list $(type_1 name_1, \dots, type_n name_n)$, this predicate will generate the output `"(+ name1+ ", "+...+ namen+ ")`, which is exactly what we need to print the actual parameter values.

The logging aspect can now employ `actualsAsString` to provide a more detailed log output; the code generated will be specific per method to which the advice is applied. Note that implementing an equivalent aspect with ASPECTJ requires the usage of runtime reflection in order to iterate over each actual parameter in a method-independent manner.

JTL AOP can be used to define not only *before*, but also *around*, *after returning* or *after throwing* advice, by renaming the original method and creating a new version which embeds a call to the original.

The following section discusses additional uses for JTL, outside of AOP, that can be reached by replacing, augmenting, or subclassing existing classes.

4.3 Templates, Mixins and Generics

Since JTL can generate code based on a given JAVA type (or list of types), it can be used to implement generic types. The `singleton` pattern below is a simple example: it is a generic that generates a SINGLETON class from a given base class. Given class, e.g., `Connection`, this predicate will generate a new class, `SingletonConnection`, with the regular singleton interface:

```

1 singleton := "public" class "Singleton"+ '?*', %[ # is T] {
2   %[public constructor ()]
3   | %2 "#T has no public zero-args constructor.";
4   last \[
5     private #T() { // No public constructor
6       private static #T instance = null;
7       public static #T getInstance() {
8         if (instance == null) instance = new #T();
9         return instance;
10    }
11  \];
12 }

```

This seemingly trivial example cannot be implemented using JAVA's generics, because those rely on erasure [6]. It takes the power of NEXTGEN, with its first-class genericity, to define such a generic type.

The JTL pattern is also superior to the C++ template approach, because the requirements presented by the class (its *concept* of the parameter) are expressed explicitly. The lack of concept specification mechanism is an acknowledged limitation of the C++ template mechanism [34]. With the JTL example above, in case the provided type argument does not include an appropriate constructor (i.e., does not match the concept), a straightforward error message is printed to `stderr` (line 3). This will be appreciated by anyone who had to cope with the error messages generated by C++ templates.

Because the generic parameter does not undergo erasure, JTL can also be used to define mixins [5]. Here is an example that implements the classic mixin `Undo` [3]:

```

undoMixin := "public" class [# is T]
  "Undoable#T extends #T" {
  %[!private void setName(String)]
  | %2 "#T has no matching setName method.";
  %[!private String getName()]
  | %2 "#T has no matching getName method.";
  all ![!private undo()]
  | %2 "Conflict with existing undo method.";
  last \[
    private String oldName;
    public void undo() { setName(oldName); }
    public void setName(String name) {
      oldName = getName();
      super.setName(name);
    }
  \];
}

```

Here, too, the pattern explicitly specifies its expectations from the type argument—including not only a list of those members that must be included, but also a list of members that must *not* be included (to prevent accidental overriding [3]).

4.4 Non-JAVA Output

There is nothing inherent in JTL that forces the generated output to be JAVA source code. Indeed, some of the most innovative uses generate non-JAVA textual output by applying JTL programs to JAVA code. This section presents a few such examples.

A classic nonfunctional concern used in aspect-oriented systems is persistence, i.e., updating a class so that it can store instances of itself in a relational database, or load instances from it. In most modern systems (such as JAVA EE v5), the mapping between classes and tables is defined using annotations. For example, here are two classes, mapped to different tables, with a foreign key relationship between them:

```
@Table class Account {
    @Id @Column long id; // Primary key
    @Column float balance;
    @ForeignKey @Column(name="OWNER_ID") Person owner;
}

@Table(name="OWNER") class Person {
    @Id @Column long id;
    @NotNull @Column String firstName;
    @NotNull @Column String lastName;
}
```

In this simplified example, the annotation `@Table` marks a class as persistent, i.e., mapped to a database table. If the `name` element is not specified, the table name defaults to the class name. Similarly, the annotation `@Column` marks a persisted field; the column name defaults to the field's name, unless the `name` element is used to specify otherwise. The special annotation `@Id` is used to mark the primary-key column.

Given classes annotated in such a manner, we can use the `generateDDL` JTL program to generate SQL DDL (Data Definition Language) statements, which can then be used to create a matching database schema:

```
generateDDL := %class "CREATE TABLE" tableName %{
    first "("; last ")"; between ",";
    %[ @Column field ] => %sqlType
    | %2 ["Unsupported field type, field" '?*'];
    columnName sqlType sqlConstraints;
%}

sqlType := %String "VARCHAR" | %integral "INTEGER" | %real "FLOAT" | %Date "DATE" |
    %boolean "ENUM('Y','N')" | %BigDecimal "DECIMAL(32,2)" | foreignKey;

sqlConstraints := [ %@NotNull "NOT NULL" | nothing ]
    [ %@Id "PRIMARY KEY" | nothing ]
    [ %@Unique "UNIQUE" | nothing ];

foreignKey := %[ field, _ is FK ] --target table/class is the field's own type
    "FOREIGN KEY REFERENCES" FK.tableName;

tableName := [ %@Table(name=TName:STRING) "#TName" ]
    [ [ %@Table() '?*' ] --Default table name = class name
    | %2 "Class is not mapped to DB table.";

columnName := [ %@Column(value=CName:STRING) "#CName" ]
    [ [ %@Column() '?*' ]; --Default column name = field name
```

Using the `first`, `last`, and `between` directives, this query generates a comma-separated list of items, one per field in the class, enclosed in parenthesis. The program also includes error checking, e.g., to detect fields with no matching SQL column type.

When applied to the two classes presented above, `generateDDL` creates the following output (pretty-printed here for easier reading):

```
CREATE TABLE Account (id INTEGER PRIMARY KEY,
    balance FLOAT,
    OWNER_ID FOREIGN KEY REFERENCES OWNER);
```

```
CREATE TABLE OWNER (id INTEGER PRIMARY KEY,
  firstName VARCHAR NOT NULL,
  lastName VARCHAR NOT NULL);
```

In much the same way, JTL can be used to generate an XML Schema or DTD specification, describing an XML file format that matches the structure of a given class.

5 Implementation Issues

This section describes how the baggage extension affects the JTL-to-DATALOG compilation process (some familiarity with DATALOG [1] is assumed) and related implementation considerations. §5.1 begins the discussion by explaining first how plain JTL is compiled to DATALOG. Then, §5.2 explains how the computation of baggage is incorporated into the process. §5.3 deals with the rather intricate issue of processing lists. Finally, §5.4 explains how the implementation of disjunction is commutative for all arguments except for baggage, and justifies this design decision.

5.1 Translating JTL into Datalog

JTL provides high-level abstraction on top of a DATALOG core. We will now briefly illustrate how JTL source code can be translated into DATALOG. The examples presented here only serve to highlight the fundamentals of the JTL to DATALOG mapping; the full details of the translation algorithm are beyond the scope of this paper. Some of the optimization techniques and algorithms used, e.g., to prevent the generation of non-terminating programs, including correctness proofs, are included in a paper by one of the current authors and others [10].

The first translation step is that of the subject variable: the subject variable in JTL is translated into a standard DATALOG variable which prepends all other actual arguments. For example, the JTL expression `p1 := abstract, extends x, x abstract;` is equivalent to this DATALOG expression:

```
p1(This) :- abstract(This), extends(This,X), abstract(X).
```

Disjunctive expressions are not as simple since DATALOG requires the introduction of a new rule for each branch of a disjunctive expression. Thus,

`p2 := public [interface | class];` is translated into:

```
p2(This) :- public(This), aux(This).
aux(This) :- interface(This).
aux(This) :- class(This).
```

The following predicate poses a greater challenge:

```
p3[T] := public extends T [T abstract | interface];
```

Here, the parameter `T` appears in the `extends` invocation and also on the left-hand side of the disjunction, but not on the right-hand side. The translation into DATALOG requires the use of a special EDB predicate, `always(X)`, which holds for every possible `x` value:

```
p3(This) :- public(This), extends(This,T), aux(This,T).
aux(This,T) :- interface(This), always(T).
aux(This,T) :- abstract(T), always(This).
```

The translation of quantifiers relies on the natural semantics of DATALOG, where every predicate invocation induces an implicit existential quantifier. For example,

```
p4 := class members: { abstract; };
```

Is equivalent to this DATALOG definition:

```
p4(This) :- class(This), members(This,M), abstract(M).
```

By using negation, we can express the universal quantifier in terms of the existential one, the negative quantifier in terms of the universal one, etc.

5.2 Calculating String Results

The output mechanism does not require the introduction of any side-effects to JTL. Rather, when compiling JTL predicates to DATALOG, we have that the string output is presented as an additional “hidden”, or implicit parameter to DATALOG queries. This parameter is used for output only. For example, the JTL predicate

`pa := public abstract; compiles to DATALOG as:`

```
pa(This,Result) :- public(This,Result1),
    abstract(This,Result2),
    concatenate(Result1,Result2,Result).
```

Thus, `Result` is the “baggage” implicit parameter that gave this mechanism its name.

Multiple output streams (§3.2) mandate the use of multiple baggage variables. The “redirection” of output from one stream to another involves changing the order in which the baggage variables are passed from one DATALOG predicate to another; redirection to a variable (the `%n>V` syntax) implies binding the JTL variable `V` to a baggage parameter.

5.3 List Processing

Many previous applications of the logic paradigm for processing software employed PROLOG rather than DATALOG. CodeQuest [20] pioneered the use of DATALOG (with stratified negation) for this task. Although PROLOG is more expressive than DATALOG because it allows function symbols in terms, it is less prone to automatic reasoning, such as deciding termination. JTL is unique in that it is DATALOG-like, but assumes an underlying infinite database; as shown by Ramakrishnan et al. [32], the move to a database with infinite relations (represented by the Extensional Database, or EDB) makes it possible to capture some of the expressive power of function symbols (such as list processing) in DATALOG.

Lists in JTL are represented using the kinds `TYPES` (a list of `TYPE` elements), `STRINGS`, etc. List processing is done using the EDB predicates `head_is`, `tail_is`, and `nil`, each of which has an overloaded version per list kind. `L.head_is[H]` holds if `H` is the first item of list `L`; `L1.tail_is[L2]` holds if `L2` equals `L1` sans the head; and `L.nil` holds if `L` is empty. Given this trio of EDB predicates, it is mundane to define predicates such as `concat` for list concatenation (see [32, ex. 7]), and others.

We can now explain how, e.g., argument list predicates are evaluated, using `args`, a primitive binary predicate. `M.args[LT]` holds if `M` is a method and `LT` (of kind `TYPES`) is its list or argument types. One may use `args` to write arbitrary recursive predicates for any desired iterative processing of the list of arguments of a method; list queries build on top of this ability. A list quantification pattern, such as

```
args: (many abstract,int,exist final)
```

is evaluated in two steps: (a) list generation; and (b) application of the quantified conditions to the list—this is achieved by searching for a disjoint partitioning of the list into sublists that satisfy the quantifiers. In this example, we search for `LT` such that `#.args[LT]` holds, and then apply the three quantifiers to it. The predicate holds if there are sublists `L1`, `L2`, and `L3`, such that `LT` is the concatenation of the three, and it holds that there is more than one `abstract` type in `L1`; `L2` has precisely one element, which matches `int`; and there is at least one `final` type in `L3`.

With list queries, there is no default quantifier; instead, a predicate expecting a list parameter is considered a quantifier, and a predicate expecting a *list element* parameter is the quantifier requiring that the respective sublist has exactly one element matching this pattern. The default generator for list queries is `args:` (compared with `members:` for set queries).

Now, the argument list pattern `()` (matching functions with no arguments) is shorthand for `args:(empty)`, while pattern `(*)` is shorthand for `args:(all true)`. Similarly, the argument list pattern `(_,String,*)` is shorthand for the more explicit pattern `args:(one true,String,all true)`.

5.4 Non-commutativity of the Disjunction Operator

The extension requires that for disjunction, output will be generated only for the first matched branch. To this end, each branch of the disjunction is considered true only if all previous branches are false; i.e., a pattern such as `p_or_a := public | abstract;` is compiled to:

```
p_or_a(This,Result) :- public(This,Result).
p_or_a(This,Result) :- !public(This,_), abstract(This,Result).
```

Note that the operation remains commutative with regard to the question of which program elements match it; the pattern `abstract | public` will match exactly the same set of elements. Commutativity is compromised only with regard to the string output, where it is undesired.

To better appreciate this design choice consider predicate `add_equals`, which unconditionally adds an `equals()` method to its implicit class argument. Then, there is a straightforward implementation of tautology `fix_equals` which only adds this method if it is not present:

```
fix_equals := has_equals declaration | add_equals;
```

(We assume a standard implementation of the obvious predicate `has_equals`.) However, this implementation will fail if `baggage` is computed commutatively. The remedy is in the more verbose version

```
fix_equals := has_equals declaration | !has_equals add_equals;
```

More generally, it was our experience that in the case of alteration between several alternatives only one output is meaningful. Our decision then saves the overhead of manual insertion of code (which is quadratically increasing in size) to ensure mutual exclusion of these alternatives. Conversely, if several alternatives are satisfied, we found no way of combining their output commutatively.

6 Output Validation

A limitation of using JTL for transformations is that it is not “type safe”, in the sense that there is no assurance that all possible outputs are valid programs in the target language.⁶ However, code generated by a JTL program is never executed directly; it must first be compiled by the target language’s compiler. Thus, the lack of output type safety in JTL will never manifest itself at runtime.

In ASPECTJ, type safety (in the sense above) is achieved by minimizing the expressiveness of its output language: complex situations, e.g., iteration over parameters of

⁶ A recent work that aims for this goal is MorphJ [21]

advised methods, are deferred to runtime and are actually implemented by reflection-based JAVA code. Thus, while ASPECTJ weaves valid JAVA bytecode, this code can actually fail at runtime due to type-safety issues. Conversely, in JTL, a similar effect is achieved by writing a predicate that iterates over the parameters of a method and generates advised code as its result (see example in §4.2). The generated code still has to undergo the standard JAVA compilation, thereby ensuring that it is well-typed, and it cannot fail at runtime for type-safety reasons (unless of course the author chooses to generate code that uses reflection).

Formally proving that a given JTL program produces only valid output in its target language is more complex, although it might be possible. Even proving that a plain procedural program produces correct SQL is known to be difficult [9, 27] and, in its most general form, undecidable. Yet the nature of string production in JTL is such that it is governed by a context-free grammar (CFG); that is, every JTL predicate also serves as a CFG of all possible results it may produce. For example, the predicate `[public | protected] static` has two possible outputs, `public static` OR `protected static`.

The problem if CFG inclusion is known to be undecidable in the general case. Still, the fact that JTL programs are CFGs can be used to generate the required formal proof in some cases. Minamide and Tozawa [28] show that it is possible (and practical) to check, for any given grammar G , whether $G \subseteq G_{\text{XML}}$ (the grammar of XML), and further, given a fixed XML Data Type Definition D , the problem of whether $G \subseteq G_D$ is also decidable. Minamide and Tozawa use this result for checking if a given PHP program produces correct XHTML (a specific XML DTD), but they rely on converting the imperative statements of PHP into a grammar specification. This conversion is approximate by nature; in contrast no such approximation is needed with JTL, and one can automatically check, for any given JTL program p and any given DTD D , whether the output of p conforms to D . The problem is more difficult, however, for target languages such of SQL or JAVA.

7 Related Work and Discussion

The work on program transformations is predated to at least D. E. Knuth’s call for “program-manipulation systems” in his famous “Structured programming with go to statements” paper [24]. Shortly afterwards, Balzer, Goldman and Wile [4] presented the concept of *transformational implementation*, where an abstract program specification is converted into an optimized, concrete program by successive transformations.

By convention, transformations in JTL have two components: *guards* (similar to a “pointcut” in the AOP terminology), which are logical predicates for deciding the applicability of a *transformer* (similar to “advices”), which is a tautology predicate in charge of output production. Examples in this paper show that JTL is an expressive tool for such output production—the transformation, or the process of aspect application, is syntax directed, much like syntax-directed code generation in compiler technology.

JTL can be categorized using Wijngaarden and Visser’s taxonomy of transformation systems [35] as a *local-input, local-output, source-driven, single-stage* system.

This work shows how a certain extent of aspect weaving can be presented as rephrasing transformations. This perspective was presented earlier by Fradet and Südholt [18], whose work focused on “aspects which can be described as static, source-to-source program transformations”. It was in fact one of the earliest attempts to an-

swer the question, “what exactly *are* aspects?”. Unlike JTL, the framework presented by Fradet and Südholt utilizes AST-based transformations, thereby offering a richer set of possible join-points, enabling the manipulation of method internals.

Lämmel [25] also represents aspects as program transformations, whereas the developers of LOGICAJ [33] go as far as claiming that “[t]he feature set of ASPECTJ can be completely mapped to a set of conditional program transformations”.⁷ LOGICAJ uses program transformations as a foundation for AOP, and in particular for extending standard AOP with generic aspects. More recently, Lopez-Herrejon et al. [26] developed an algebraic model that relates aspects to program transformations.

The ELIDE system for Explicit Programming [8] defines *modifiers* that are placed, somewhat like annotations, in JAVA code; programs associated with these modifiers can then change the JAVA code in various ways, including the generation of new methods and classes or the insertion of code before/after methods. By using queries that match standard Java annotations, JTL transformations can be used to a similar effect.

Unlike JTL, ELIDE handlers use JAVA-like code to modify the base JAVA code; yet similarly to JTL, ELIDE’s code can include multi-line strings (enclosed in `#{ ... }`) and has an “escape” syntax for quoting variables inside such strings.

The Stratego system [36] is a generic term rewriting tool. As such it is useful in a wider range of applications. By focusing on the domain of Java programs, JTL sports a nicer and more intuitive syntax, thus making it more user friendly.

JTL is not the first system to use logic-based program transformation for weaving aspects. Indeed, De Volder and D’Hondt’s [15] coin the term *aspect-oriented logic meta programming* (AOLMP) to describe logic programs that reason about aspect declarations. The system they present is based on TYRUBA [14], a simplified variant of PROLOG with special devices for manipulating JAVA code. Whereas JTL presents an open-ended and untamed system, De Volder and D’Hondt’s system presents a very orderly alternative, where output is generated using quoted code blocks.

We therefore find that, compared to other AOP-by-transformation systems, JTL is limited in the kind of transformations it can apply for weaving aspects, and the level of reasoning about aspects that it provides—which is why we view it as a “quick-and-dirty” AOP language. The windfall, however, is that program transformation in JTL is not limited to AOP alone, as evident from some of the examples provided in this paper—the generation of stub classes from interfaces, the generation of SQL DDL to match classes, the definition of generic classes, etc.

Acknowledgments. We thank Andrew Black for his meticulous review and insightful comments.

References

1. M. Ajtai and Y. Gurevich. Datalog vs. First-Order Logic. FOCS’89.
2. E. Allen, J. Bannet, and R. Cartwright. A first-class approach to genericity. OOPSLA’03.
3. D. Ancona, G. Lagorio, and E. Zucca. Jam—designing a Java extension with mixins. *ACM Trans. Prog. Lang. Syst.*, 25(5):641–712, 2003.
4. R. Balzer, N. M. Goldman, and D. S. Wile. On the transformational implementation approach to programming. ICSE’76.

⁷ <http://roots.iai.uni-bonn.de/research/tailor/aop>

5. G. Bracha and W. R. Cook. Mixin-based inheritance. OOPSLA'90.
6. G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to the Java programming language. OOPSLA'98.
7. J. Brichau, A. Kellens, K. Gybels, K. Mens, R. Hirschfeld, and T. D'Hondt. Application-specific models and pointcuts using a logic meta language. ESUG'06.
8. A. Bryant *et al.* Explicit programming. AOSD'02.
9. A. S. Christensen, A. Møller, and M. I. Schwartzbach. Precise analysis of string expressions. SAS'03.
10. S. Cohen, J. Y. Gil, and E. Zarivach. Datalog programs over infinite databases, revisited. DBPL'07.
11. T. Cohen. *Applying Aspect-Oriented Software Development to Middleware Frameworks*. PhD thesis, The Technion — Israel Institute of Technology, 2007.
12. T. Cohen, J. Y. Gil, and I. Maman. JTL—the Java Tools Language. OOPSLA'06.
13. T. Cohen, J. Y. Gil, and I. Maman. JTL and the annoying subtleties of precise μ -pattern definitions. *Int. Workshop on Design Pattern Detection for Rev. Eng.*, 2006.
14. K. De Volder. *Type-Oriented Logic Meta Programming*. PhD thesis, Vrije Universiteit Brussel, 1998.
15. K. De Volder and T. D'Hondt. Aspect-oriented logic meta programming. *Int. Conf. on Reflection*, 1999.
16. E. W. Dijkstra. Guarded commands, non-determinacy and a calculus for the derivation of programs. In F. L. Bauer and K. Samelson, editors, *Lang. Hierarchies and Interfaces*, 1975.
17. R. E. Filman and K. Havelund. Realizing aspects by transforming for events. ASE'02.
18. P. Fradet and M. Südholt. An aspect language for robust programming. ECOOP'99 workshops.
19. K. Gybels and A. Kellens. An experiment in using inductive logic programming to uncover pointcuts. *European Interactive Workshop on Aspects in Software*, 2004.
20. E. Hajiyev, M. Verbaere, and O. de Moor. CodeQuest. ECOOP'06.
21. S. S. Huang and Y. Smaragdakis. Class Morphing: Expressive and Safe Static Reflection. PLDI'08.
22. D. Janzen and K. De Volder. Navigating and querying code without getting lost. AOSD'03.
23. G. Kiczales *et al.* Aspect-Oriented Programming. ECOOP'97.
24. D. E. Knuth. Structured programming with goto statements. *ACM Comp. Surv.*, 6(4), 1974.
25. R. Lammel. Declarative aspect-oriented programming. In *Partial Evaluation and Semantic-Based Program Manipulation*, 1999.
26. R. Lopez-Herrejon *et al.* A disciplined approach to aspect composition. PEPM'06.
27. E. Meijer, B. Beckman, and G. Bierman. LINQ. SIGMOD'06.
28. Y. Minamide and A. Tozawa. XML validation for context free grammars. APLAS'06.
29. N. Minsky. Towards alias-free pointers. ECOOP'96.
30. N. Minsky and J. Leichter. Law-governed Linda as a coord. model. ECOOP'94 workshops.
31. K. Ostermann *et al.* Expressive pointcuts for increased modularity. ECOOP'05.
32. R. Ramakrishnan *et al.* Safety of recursive Horn clauses with infinite relations. PODS'87.
33. T. Rho *et al.* LogicAJ home page, 2006. <http://roots.iai.uni-bonn.de/research/logicaj/>.
34. B. Stroustrup and G. D. Reis. Concepts—design choices for template argument checking. ISO/IEC JTC1/SC22/WG21 no. 1522, 2003.
35. J. van Wijngaarden and E. Visser. Program transformation mechanics. Technical Report UU-CS-2003-048, Utrecht University, 2003.
36. E. Visser. *Stratego. Rewriting Techniques and Applications*, 2001.
37. E. Visser. A survey of strategies in program transformation systems. *ENTCS*, 57, 2001.
38. M. M. Zloof. Query By Example. In *Proc. of the Nat. Comp. Conf.*, 1975.