Behavioral Similarity Matching using Concrete Source Code Templates in Logic Queries

Coen De Roover *
Theo D'Hondt
Programming Technology Lab
Vrije Universiteit Brussel, Belgium
{cderoove,tjdhondt}@vub.ac.be

Johan Brichau
Département d'Ingénierie Informatique
Université catholique de Louvain, Belgium
johan.brichau@uclouvain.be

Carlos Noguera
Laurence Duchien
INRIA - Futurs, Jacquard Project
LIFL, UMR CNRS 8022, Equipe GOAL - Lille, France
{noguera,duchien}@lifl.fr

Abstract

Program query languages and pattern-detection techniques are an essential part of program analysis and manipulation systems. Queries and patterns permit the identification of the parts of interest in a program's implementation through a representation dedicated to the intent of the system (e.g. call-graphs to detect behavioral flaws, abstract syntax trees for transformations, concrete source code to verify programming conventions, etc). This requires that developers understand and manage all the different representations and techniques in order to detect various patterns of interest. To alleviate this overhead, we present a logic-based language that allows the program's implementation to be queried using concrete source code templates. The queries are matched against a combination of structural and behavioral program representations, including call-graphs, points-to analysis results and abstract syntax trees. The result of our approach is that developers can detect patterns in the queried program using source code excerpts (embedded in logic queries) which act as prototypical samples of the structure and behavior they intend to match.

Categories and Subject Descriptors F.3.2 [*Semantics of Programming Languages*]: Program Analysis; D.2.2 [*Software Engineering*]: Coding Tools and Techniques; D.1 [*Programming Techniques*]: Logic Programming

General Terms Design, Languages, Algorithms, Verification

Keywords Source code templates, program querying, pattern detection, logic meta programming, program analysis, program validation.

1. Introduction

Modern software development environments are generally equipped with advanced code analysis and verification tools in support of software quality management. Such tools often verify coding styles and conventions, detect possible bugs and bad smells [13] or produce a set of metrics about the code [15]. In most cases, the full potential of such tools can only be realized if developers themselves can implement which styles, conventions, bugs, bad smells, etc. are to be detected and verified by these tools. Coding conventions and styles, for example, are often specific to a project development context. Therefore, a tool that checks coding conventions must provide means to both express the conventions and select program elements that need to adhere. Similarly, a tool that checks for possible run-time errors should provide a means to detect application-specific errors (for example, in case of an API of which the functions must be called in a particular order).

Program query and pattern languages have already proven valuable for the detection of user-defined conventions [18], code smells, errors [17, 6], design patterns [25], best-practice patterns [5], etc. In general, these meta-programming systems allow implementation patterns to be detected by querying or matching a particular representation of the program under investigation. This representation is defined by a *meta model* and it is most often fine-tuned for a particular purpose. Systems that focus on the verification of coding conventions and styles generally only require a structural meta model, conveying only the static structure of the program. In contrast, the detection of run-time errors requires a behavioral meta model that exposes run-time behavior of the program, such as its control flow, call-graph, etc. Nevertheless, many pattern detection problems can benefit from the availability of structural as well as behavioral information about the application. A general-purpose program investigation tool would thus need to provide different representations, according to different meta models, of the application. However, such an approach will typically hamper the adoption by developers since they must have detailed knowledge about different meta models and the implementation of the queries can easily become complex.

To reconcile the need for different meta models with the ease of specification of queries and pattern detection problems, we have extended a logic program query language with a template-based pattern specification mechanism. This template-based meta-programming system allows the detection of patterns in software that require diverse representations of the code (i.e. structural as well as behavioral representations). Developers can specify templates that represent prototypical implementations of the patterns they wish to detect. The system then matches these templates on the static source code structure of the system as well as on a call-graph and a points-to analysis of the code. In particular, the call-graph and points-to analysis-based representations allow to conceal the matching of a sequence of statements and expressions into the matching on control-flow and run-time values respectively. As a

* Ph.D. scholarship funded by the "Institute for the Promotion of Innovation through Science and Technology in Flanders" (IWT Vlaanderen).

PEPM 2007 January 15-16, 2007, Nice, France.

```
class Y {
  private X var;
  public X getVar { return var; }
  public void setVar(X val) { var = val; }
}
```

Figure 1. Prototypical getter and setter method best practice pattern implementations in Java.

result, the system allows the detection of structural code patterns as well as run-time errors in the program and provides a uniform and simple pattern specification mechanism to express both. Furthermore, the templates can be embedded in logic queries, thereby still enabling developers to compose different patterns using logic operators and resort to pure query programs whenever the need arises. Most importantly, developers are not required to understand the intricate details of the structural and behavioral representations of the program that are required to perform the pattern detection.

Outline of the paper Section 2 starts the exposition of our approach to pattern detection with the prime motivations to reconcile a program query language with concrete source code templates. How these templates are resolved against a wide range of program representations is described in Section 3. Section 4 demonstrates how different variants of some interesting patterns can be detected using templates in logic queries. Related work will be reviewed in Section 5 and we will conclude our discussion in Section 6.

2. A Logic Pattern Detection Language

A substantial amount of the software development process pertains to the application of diverse kinds of patterns that describe already known solutions to particular implementation problems. Design patterns, coding conventions, protocols, best-practice patterns and even anti-patterns such as bad code smells or coding habits are but a few kinds of patterns that represent a vast body of knowledge available to the software engineer. Although there exist many possible descriptions of such patterns, developers primarily think of them by means of examples or prototypical implementations that focus on the essential parts of the pattern. For example, Figure 1 contains a prototypical implementation of the well-known *accessor methods*[1] pattern. When applying or discovering such a pattern in an application, developers tend to do so in terms of the example implementation they already know. However, when they need to automate the pattern-detection process, they need to convey their declarative knowledge into operational queries or other intricate descriptions that cope with the various possible instantiations and variations of the same pattern. This is because it is expected that a query discovers implementations of the pattern that are not formulated exactly as the prototype, but that stay true to the spirit of the prototype. In the case of the getter prototype, the query engine should also discover getter method instances that apply lazy-instantiation, for example, as long as the method still is a getter. Furthermore, an additional burden is often caused by the intricate details of the query language and the diverse representations of the program that are required to detect diverse kinds of patterns.

We can lessen this burden imposed on developers by bringing the description of these queries closer to the prototypical example implementations of the queried pattern. In doing so, it is however important not to lose the expressive power of query languages that allows the detection of the different variations of the pattern. In the remainder of this section, we present our approach that reconciles the advantages of query languages and diverse representations of

[1] Also know as getter and setter methods.

```
1  isAncestorOf(?root, ?directSubclass) if
2    isSubClassOf(?directSubclass, ?root).

3  isAncestorOf(?root, ?indirectSubclass) if
4    isSubClassOf(?indirectSubclass, ?parent),
5    isAncestorOf(?root, ?parent).

6  if isAncestorOf(java.lang.Object, FooBar)
7  if isAncestorOf(?superclass, FooBar)
8  if isAncestorOf(java.lang.Object, ?subclass)
9  if isAncestorOf(?superclass, ?subclass)
```

Figure 2. Rules describing the ancestor-relation between two classes in the logic programming paradigm.

the same program with pattern detection based on prototypical example implementations.

2.1 A Logic Program Query Language

We rely on a logic programming language [9] to query the object-oriented program under investigation. In such a logic meta programming approach, descriptions of implementation patterns are expressed as logic conditions over the object-oriented program's elements. Detecting the presence of an implementation pattern therefore amounts to initiating a search for program elements adhering to these conditions. In logic programming, such conditions are grouped into reusable logic rules while the search for solutions is initiated by launching a logic query.

The use of a logic programming language for meta programming purposes has several well-established advantages [2, 25]. In imperative programming languages, programmers specify exactly *how* the solution to a problem is to be found using step-by-step algorithmic descriptions. In contrast, logic programming languages allow the problem itself to be specified. The programming language will find a solution on its own, relying on a specific problem solving strategy employed by the language.

To illustrate the practical difference between these two programming paradigms for meta-programming purposes, consider the problem of finding all ancestors of a particular class. As shown in Figure 2, it suffices to describe[2] what it means for one class to be an ancestor of another class in the logic programming paradigm.

The first rule expresses that a class *?root* is the ancestor of a class *?directSubclass* when the latter is a subclass of the ancestor class. The second rule expresses that a *?root* class is also the ancestor of a subclass of a class it is already the ancestor of. To find all ancestor of a class `FooBar`, one can launch the logic query on line 9 and the logic programming system will present us bindings for the *?superclass* logic variable for which the `isAncestorOf/2` relation holds.

In the imperative paradigm, we can easily find all ancestors of a class as well — this time through an algorithm which retrieves superclasses, for instance using the Java code shown in Figure 3. Solving meta-programming problems using a logic programming language however has some distinct benefits. First of all, logic rules describe relations between their arguments in a declarative instead of an operational manner. A single logic rule can therefore be used to verify whether there is an ancestor relation between two classes (line 8), to find all superclasses of a given class (line 9), but also to find all subclasses of a given class (line 10), or even to find all class pairs for which there exists an ancestor relation (line 11). As

[2] The syntax of the SOUL logic program query language, which has some specialized features to support meta-programming purposes, slightly differs from the one of the prototypical logic programming language Prolog. Logic variables start with a question mark, standalone question marks denote anonymous variables, while lists are denoted as: `<1,?number,?,2>`.

```
Class subclass;
LinkedList superclasses = new LinkedList();
Class superclass = subclass.getSuperclass();
while (superclass != null) {
  superclasses.add(superclass);
  subclass = superclass;
  superclass = subclass.getSuperclass();
}
```

Figure 3. Algorithm gathering the superclasses of a class in the imperative programming paradigm.

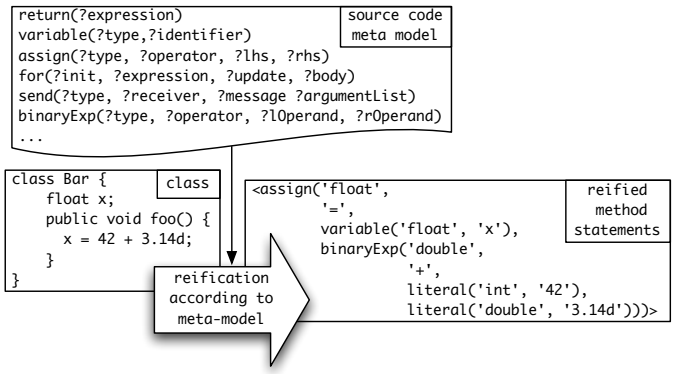

Figure 4. Logic representation of a typical Java method according to a source code meta model.

one has to specify a solution strategy instead of an expressive description of the problem itself, one has to write different algorithms for each of these queries in the imperative paradigm. Logic programming languages are, furthermore, extremely suited to program querying purposes thanks to their advanced pattern matching abilities, built-in support for non-determinism, logic connectives and powerful programming concepts such as recursion and backtracking.

Over the years, the logic meta programming approach has been applied successfully to a variety of problems in object-oriented software engineering. Some examples include reasoning about object-oriented design [25] and enforcing implementation patterns [18].

2.2 Detecting Implementation Patterns

To support the use of the `isAncestorOf/2` predicate in logic queries, a logic program query language has to represent the object-oriented program as logic facts according to a basic *structural* meta model. These facts state the classes and methods present in the program, as well as the basic relations between them such as one class being the subclass of another —the latter being indicated by the `isSubClassOf/2` predicate. However, in order to describe and detect implementation patterns, a more fine-grained meta model is needed. To detect the accessor method or array-walk implementation patterns, knowledge about the statements in each method's body is required. For Java programs, such information is for instance provided by the Irish *source code* meta model presented in [8]. As illustrated by Figure 4, it reifies the depicted method's body as a singleton list containing a logic functor of the form `assign(?type, ?operator, ?lhs, ?rhs)` which represents the method's assignment statement.

Using the information offered by this source code meta model, we can attempt to describe and detect concrete instances of the getter method implementation pattern. The logic rule shown in

```
1 getterMethod(?class, ?method, ?field) if
2   isMethodInClass(?method, ?class),
3   fieldInClassChain(?field, ?class),
4   fieldName(?field, ?fname),
5   methodStatements(?method, ?slist),
6   ?slist = <return(variable(?vtype, ?fname))>)
```

Figure 5. Logic rule for the the getter method implementation pattern.

Figure 5 expresses what it means for a method *?method* to be a getter method for a field *?field* in a class *?class*. The first two lines state that a getter method *?method* needs to be part of a class *?class* in whose hierarchy a field *?field* is defined. In addition, the method's reified statement list is required to match the one of the prototypical getter method implementation shown in Figure 1. It has to be a list containing a single return statement of which the expression part matches a variable named exactly after the field's name *?fname*.

The multi-directional nature of rules in the logic programming paradigm allows for the `getterMethod/3` predicate to be used in queries verifying whether a method is a getter method as well as in queries detecting all getter methods in the application as a whole, a specific class or for a specific field. Such use of a logic program query language to describe and detect implementation patterns is however not without its problems.

First of all, we have lost the highly desirable separation between a problem description and an algorithm that searches for solutions to this problem. It is easy to imagine a real-life getter method which cannot be detected using this rule. A getter method might for instance log a message to a file before returning the field it is protecting. Since its statement list won't faithfully match the one of the prototypical getter method, it won't be recognized as such. One ad-hoc solution to this problem might be to alter the logic rule to traverse the method's parse tree in search for the correct return statement. Be that as it may, before long end-users will no longer be detecting implementation patterns using a declarative description, but in an operational manner describing different techniques to search for implementation variants of the pattern —effectively negating our prime motivation to use a logic program query language.

Moreover, while the information provided by the structural meta model (e.g. `isSubClassOf/2`, `isMethodInClass/2`, ...) was reasonably straightforward to use, this is no longer true for the source code meta model. The logic representation of the concrete syntax elements of a programming language is often more complex than the elements themselves. Users already familiar with a programming language, need to learn its logic reification in order to describe and detect implementation patterns. This represents a major hurdle to the system's widespread adoption. In fact, the logic rule in Figure 5 is an arguably less declarative description of the getter method pattern than its prototypical implementation outlined in Figure 1.

Finally, a base program can be represented according to multiple candidate meta models. Which meta model offers the most efficacious representation usually depends on the actual implementation patterns that are to be detected. Often a combination of the information from multiple meta models is needed. Returning to our running example, imagine a getter method in which the expression part of its return statement is not a variable named after the field the method is protecting, but rather a complex expression whose run-time value matches the value of the field. In order to correctly recognize this concrete instance of the getter method pattern, we require information about the run-time behavior of the program. Such behavioral information could be offered either by a meta model obtaining its information through a dynamic or through a static anal-

ysis of the program under investigation. As the former extracts its information from an execution trace, it is precise but only valid for one of many possible program executions. As the latter extracts its information from a conservative approximation of all possible executions, it is in contrast often less precise but always valid. However, most users are not acquainted with the specific manner in which this behavioral information is extracted, nor with the way it is presented as logic facts. Moreover, previous logic meta programming experiments with a source code meta model [8, 18], static [5] or dynamic analysis behavioral meta model [6, 7], have shown that the nature of the meta model is of great influence on how and even if a pattern can be expressed in a logic rule. This requires developers to understand the different program representations and even use different strategies to describe and detect a wide range of implementation patterns.

To summarize this section, we have identified a number of desirable properties for a logic pattern query language, which aren't exhibited by a plain logic program query language. First of all, the use of a plain logic program query language often results in an operational definition of the search for a pattern rather than the desired declarative specification of its essence. Secondly, this specification should resemble the pattern's prototypical source code implementation familiar to developers in stead of unfamiliar logic predicates which moreover depend on one specific representation of the base program. Thirdly, source code pattern specifications need to integrate seamlessly with the logic program query language. As such, they should support the language's multi-directionality, nondeterminism and unification of logic variables. In addition, it must be possible to compose specifications using the usual logic connectives `and`, `or`, and `not`. Finally, given a declarative specification of a sought pattern, the pattern query language should employ a search strategy that recognizes concrete instances of the pattern, possibly against multiple base program representations.

2.3 Source Code Templates

In order to support the declarative description and efficacious detection of implementation patterns, we have extended the SOUL logic program query language with a source code template construct. These templates support the declarative specification of implementation patterns in the syntax of the base program language. An implementation pattern is thus described using its prototypical source code implementation with provisions for variability. Source code templates resemble concrete source code with "cutouts" demarcated by logic variables. We support five different types of source code templates: statements, expressions, field declarations, method declarations and class declarations. We will briefly introduce each template type using an example query.

Statement Templates A number of logic source code template constructs for Java statements are depicted in Figure 6. Each template comprises a single-argument predicate `jtStatement/1`[3], followed by a sequence of concrete syntax elements delimited by braces. These are exactly those elements recognized by the `Statement` production rule in the Java Language Specification's grammar [10]. Source code templates can be used in the logic program query language anywhere a logic condition is allowed: e.g. in the body of logic rules or in a logic query. While the next section will detail how source code templates are resolved, it suffices to say for now that a template's concrete syntax elements are matched against the program under investigation and that any matching program element will be bound to —or more precisely, unified with— the argument in the template's predicate. When the logic programming language is asked for all solutions to the first logic query in

[3] The functor of the jtStatement/1 predicate stems from Java Template Statement.

```
1  if jtStatement(?statement){ var = val; }

2  if jtStatement(?statement){ return ?e; }
3  if jtStatement(?statement){ return ?e; },
4     isReturn(?statement), isExpression(?e)
5  if jtStatement(?statement){ ?x = (?type) ?e; }

6  if jtStatement(?s){ if(?cond) ?t;  else ?e; },
7     or(isBlock(?t), isReturn(?t))
8  if jtStatement(?s){ if(?cond) {?t;}  else ?e; }

9  if jtStatement(?statement){ ?statement; }
10 if jtStatement(?statement){ ?s; },
11    equals(?s,?statement)
```

Figure 6. Some example logic queries involving source code templates for statements.

Figure 6, it will present bindings for the logic variable *?statement* matching the Java source code between the template's braces. In other words, through the template's predicate argument we will be presented all Java statements assigning a variable `val` to a variable `var`.

The second example query illustrates that logic variables can be used as placeholders for productions originating from a non-terminal in the `Statement` grammar production rule. Solutions to this query will have a return statement bound to the *?statement* logic variable, while the expression-part of this statement will be bound to the *?e* logic variable. The logic query on lines 4–5 comprises the same template and two additional logic conditions which explicitly check the bindings for the template's variables. These conditions are however redundant as the constraints on the variable bindings are already implicitly implied by the template itself. Line 6 has another example of a source code template containing logic variables: upon evaluation, *?statement* will be bound to any assignment statement assigning to a left hand side *?x* the value of a cast of an expression *?e* to a type *?type*. The queries in lines 8–10 involve `if`-statement templates. Given a program containing the statements `if(true) return true; else return false;` and `if(true) {return 1;} else return 0;`, it is interesting to note the difference between the solutions given for the first and second query. The results for the first query will contain both `if`-statements, binding *?t* either to the statement `return true;` or to the block statement. In contrast, the results for the second query won't include the first `if`-statement, binding *?t* only to the statement `return 1;`. Finally, the query on line 12 illustrates that a logic variable followed by a semicolon is allowed in lieu of concrete syntax elements as well. It will match any statement in the program under investigation. As the query on lines 13–14 demonstrates, this logic variable will always equal the argument of the statement template's predicate.

Expression Templates Figure 7 lists a number of logic queries involving source code templates for Java expressions. Once more, the argument of the `jtExpression/1` predicate will be bound to those expressions from the actual base program matching the concrete syntax elements between the template's braces. While these elements must adhere to the `Expression` production rule in the Java grammar, logic variables are allowed as placeholders for productions originating from a non-terminal in the production rule. As the first example query illustrates, a logic variable is also allowed as the single element between a template's source code braces. In this case, the variable *?expression* will match any expression in the program under investigation. The logic queries on lines 3–7 all involve source code templates matching cast expressions. The first query will match any cast to the `Object` class, binding the logic variable *?e* to the expression whose value is being cast. The second

```
1  if jtExpression(?expression){ ?expression }

2  if jtExpression(?cast){ (java.lang.Object) ?expression }
3  if class(?type),
4     jtExpression(?expression){ (?type) ?e }
5  if jtExpression(?expression){ (?type) ?e },
6     jtFieldDeclaration(?d){ ?type ?name; }

7  if jtExpression(?expression){ ?x.?f }
8  if jtExpression(?base){ this },
9     jtExpression(?expression){ ?base.?f }

10 if jtExpression(?e){ new java.lang.Integer(?argument) }
11 if jtClassDeclaration(?c){ class Integer {} },
12    jtExpression(?e){ new ?c(?argumentList) }

13 if jtExpression(?e){ this.?selector(?argumentList) }
14 if jtExpression(?e){ ?receiver.?selector(?a1, ?a2) }
15 if jtExpression(?e){ ?x.?selector(?arg1, (?type) ?e2) }
```

Figure 7. Some example logic queries involving source code templates for expressions.

query recognizes only casts to a class found in the base program's source code. For this it relies on the `class/1` predicate from the structural meta model to restrict the template's *?type* variable. The query on lines 6–7 will in contrast only report casts to a type of a field declaration in the base program. The logic variable *?expression* in the query on line 9 will be bound to field access expressions whose base corresponds to the value of the variable *?x* and field corresponds to the value of the variable *?f*. Likewise, the query on lines 10–11 will report only accesses to expressions recognized by the template on line 10 as matching the `this` special variable. In solutions to both of the queries on lines 13–15, the logic variable *?e* will always be bound to expressions creating an instance of the `Integer` class. However, the value of the *?argument* variable in solutions to the first query will match the instance creation expression's single argument expression. The value of the *?argumentList* variable will however be a singleton list containing this argument expression. The template differentiates between both cases on the basis of a `List`-suffix naming convention for logic variables. Finally, lines 17–19 contain examples of templates matching method invocation expressions. The first template matches all `this`-sends of a message with name *?selector* and argument list *?argumentList*. The second template matches only two-argument message sends of *?selector* to any receiving expression *?receiver*, binding the first argument expression to the logic variable *?a1* and the second argument expression to the logic variable *?a2*. The template on the last line will match any two-argument sends where the second argument expression is a cast of an expression *?e2* to a type *?type*.

Declaration Templates Figure 8 concludes our discussion on the syntax of the new logic source code template construct. It lists a number of example queries involving the three remaining types of templates we haven't discussed yet: class declarations, field declarations and method declarations. In solutions to the first logic query, the logic variable *?c* will be bound to the `java.lang.Object` class. Solutions to the second query will comprise all class *?c* - superclass *?super* pairs in the queried base program. The query on line 4 illustrates a property of the logic variables in source code templates we haven't encountered before: they might function as placeholders for productions from multiple non-terminals in a production rule. In this case, the logic variable *?member* in the body of a class *?c* might represent a field declaration or a method declaration or even an inner class. The source code template in the query on line 5 is an example of a "recursive" template. It will match any class having a public field *?f* of its own type. The query on line 7

```
1  if jtClassDeclaration(?c){ class Object {} }
2  if jtClassDeclaration(?c){ class ?cn extends ?super {} }

3  if jtClassDeclaration(?c){ class ?cn { ?member } }
4  if jtClassDeclaration(?c){ class ?cn { public ?c ?f; } }

5  if jtFieldDeclaration(?f){ public Integer foo; },
6     jtClassDeclaration(?c){ class ?cn { ?f } }

7  if jtMethodDeclaration(?m){ public int ?name() { ?s;} }
8  if jtMethodDeclaration(?m){ public ?type ?name() {
9                                  ?x = ?y;
10                                 return e;
11                              }
12    }
```

Figure 8. Some example logic queries involving source code templates for class and class member declarations.

will match any class *?c* with a field *?f* matching the field `public Integer foo;`. Finally, the method declaration template on line 10 matches any method named *?name* containing a statement *?s*. The last method declaration template matches any method of type *?type* named *?name* containing an assignment statement followed by a return of an expression *?e*.

3. Resolving Source Code Template Queries

Now the syntax of the logic source code template construct has been established, it is time to detail the way these templates are matched against the program under investigation.

As pointed out in section 2.2, one of the main objectives for our logic pattern query language was to support the description of an implementation pattern in a declarative manner using its prototypical implementation. To this end, we introduced logic source code templates as new syntactic elements in the logic program query language. Solutions to a logic query involving such a template consist of bindings for the template's predicate argument adhering to the constraints imposed by the template's source code. Given a source code template describing a pattern's prototypical implementation, the logic language should however also recognize as many of the pattern's implementation variants as possible. The constraints a template imposes on the bindings for its predicate argument are therefore not solely syntactic in nature. Instead, these constraints demand that a positive match gives rise to run-time behavior that is comparable to the behavior the template's source code implements. This way, although logic templates contain concrete source code, they are able to capture the intention of the code.

3.1 Non-behavioral Constraints

We will start our exposition with the non-behavioral constraints imposed by a template on its predicate argument. These are constraints which rely only on the structural and source code meta models introduced in section 2.1. Given our flexible program querying goal, we have made the deliberate decision to include only constraints for those concrete syntax elements that are explicitly listed in a template's source code. This design decision accounts for the previously highlighted difference between the solutions for the `if`-template queries on lines 8–10 of Figure 6. As a block statement is listed among the source code in the second query's template, it is also required in bindings for its predicate argument. The same design decision explains why a solution for the first class declaration template in Figure 8 binds the variable *?c* to the class `java.lang.Object` even though the template's class body is empty. Likewise, an empty body in a method declaration template is interpreted as if no constraints are desired on the

statements in the matching methods[4]. In contrast, if a statement is present in a method declaration template's body, it should also occur in bindings for its predicate argument. Whenever the template body comprises multiple statements, some additional constraints enforce their ordering. As we will see later on, these sequencing constraints are however behavioral in nature.

3.2 Behavioral Constraints Imposed by Expressions

Section 2.2 used a particular implementation variant of the getter method pattern to point out the necessity of including behavioral information in a query language's search strategy. In this implementation, the return statement's expression is not just the name of the protected field, but rather an arbitrary expression whose run-time value coincides with the value of the field. Detecting such a variant requires information about the run-time values to which expressions evaluate.

A behavioral meta model As this run-time information should be valid for every possible run of the program under investigation, an analysis has to introduce some approximations in the information it computes. Informally, many static analyses can be regarded as if they were executing the analysed program with abstract descriptions of the concrete values that appear during an ordinary program execution. To make this idea somewhat more tangible, an integer expression's parity can for instance be used as an abstract description of the set of it's possible run-time values. Of course, parity abstract value descriptions are of little use when run-time information is needed to resolve general-purpose template queries.

As run-time interactions between objects shape a program's overall behavior in the object-oriented paradigm, statically-obtained knowledge about the objects an expression might evaluate to is crucial in resolving template queries about this program. This knowledge can be derived through a points-to analysis [21]. Such a static analysis computes at compile-time the set of all heap objects a reference might point to during an execution of the program. We will use these points-to sets as abstract value descriptions for reference expressions in the program's behavioral meta model.

To obtain points-to sets, we rely on the Spark [16] toolkit of the Soot Java Optimization Framework. It implements a conservative flow-insensitive, context-insensitive points-to analysis for Java. Although flow-insensitiveness and context-insensitiveness are one of the major sources of imprecision in the analysis of object-oriented programs, they also guarantee a reasonably efficient computation [12] —whereas the precision of the analysis results has until now been sufficient for our purposes.

Resolving expression templates While the above describes the nature of the necessary behavioral information, we have not yet detailed how this information is involved in the resolution of expression templates. After all, logic template constructs contain a mixture of concrete source code and logic variables describing a pattern's prototypical implementation. For the sake of simplicity, we'll concentrate on the resolution of cast expression templates. Other expression templates are resolved analogously.

Both the type and the subexpression part of the code in a cast expression's template impose constraints on the bindings for the template's predicate argument. A first constraint demands this argument to be bound to a cast in the program under investigation. A second constraint requires the compatibility of the type the template's subexpression is cast to and the type the program's subexpression is cast to. As these constraints are still exclusively syntactic, they can be verified against the source code meta model.

[4] At this point, it should be noted that the design of our template language is still a work in progress. In the future, we would for instance like to include features to query the base program for methods with an empty body.

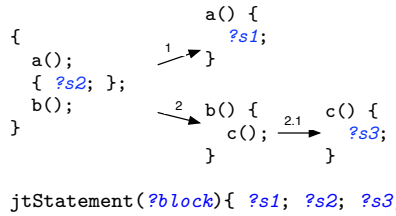

Figure 9. Multiple layers and kinds of indirection for the containment and ordering constraint to take into account.

This is however no longer the case for the constraint imposed by the template's subexpression. This constraint does not demand the syntactic equivalence of both cast subexpressions, but demands the abstract descriptions of their possible run-time values to be compatible instead. With points-to sets being the designated abstract value descriptions, this constraint amounts to determining whether both intersect. Put differently, although the casts' subexpressions might deviate syntax-wise, the constraint isn't violated in case the behavioral meta model indicates they might evaluate to overlapping sets of objects at run-time. However, as templates can contain logic variables as well, in reality there's a full-fledged unification procedure behind this constraint [5].

The same constraint-based resolution applies to expressions within a statement template. This way, logic templates resolve to base program elements implementing similar behavior. The following query highlights this property of our logic pattern query language:

```
if jtStatement(?s1){ return ?expression; },
   jtStatement(?s2){ return ?expression; },
   differs(?s1, ?s2)
```

It resolves to the set of all return statements *?s1* and *?s2* that syntactically differ, but might return overlapping sets of objects at run-time. The statements `return foo;` and `return this.getFoo();` represent possible bindings for the *?s1* and *?s2* logic variables. Any complicated expression is allowed for the second return statement's expression, provided it possibly evaluates during a program execution to the first return statement's expression. The second template's *?expression* variable effectively introduces a constraint on the possible values returned by any matching return statement.

3.3 Behavioral Constraints Imposed by Statement Sequences

In the previous section, we have briefly described the resolution of a logic template for an individual statement and the expressions it contains. However, two additional constraints need to be imposed on the resolution of a block statement or method body template. The resolution for each individual statement in the template must in some way be contained by the resolution of the template itself, i.e. the value bound to its predicate argument. This is enforced by a *containment* constraint. Moreover, the resolved statements must be executed in an order compatible with the order specified in the template. This is enforced by an *ordering* constraint. As we will see, the validation of these constraints in a program querying setting calls for yet another behavioral program meta model.

Figure 9 illustrates this need using a template query for a block statement *?block* containing three consecutive statements *?s1*, *?s2* and *?s3*. In order to detect as many implementation variants as possible, the containment constraint should take multiple layers of possible indirections into account. Depicted is a matching block statement wherein statement *?s2* resides in another block statement and wherein statement *?s3* is amongst the statements in a method called from a method called from within the matching block statement.

Statements can thus be contained within any other statement reachable from the template's *?block* binding. In addition to these indirections, the ordering constraint has to take into account the possibility of other statements between the resolved *?s1*, *?s2* and *?s3* statements —recalling our design decision to impose constraints only for the concrete syntax elements explicitly specified in a template.

Another behavioral meta model Accommodating the needs of the containment constraint, the second behavioral program meta model offers information about the methods reachable from a particular source statement. This information corresponds to all target methods within the application found on a transitive call chain starting at the source statement. To accommodate the needs of the ordering constraint, every edge along this call chain has a number representing its position in the chain. Statement *?s3* can be reached taking the first invocation in the method found by following the second invocation in the *?block* statement.

This kind of information can be obtained statically through a call graph analysis. For this, we are once more relying on the Soot Java Optimization Framework. Since it already had to perform a points-to analysis for our previous behavioral meta model, it is moreover able to compute a reasonably precise call graph for each polymorphic call site by pruning method invocation candidates based on the site's points-to set. Less precise analyses have to resort to elimination based on the call site's declared type.

3.4 An Open Implementation

To conclude our exposition of the template constructs, we think it is useful to mention that their entire resolution strategy can be adapted in a straightforward way. Each of the templates is actually translated automatically to a logic query expressing the constraints specified in the template. This translation process is again executed by a logic program which can be changed to alter the semantics of the source code templates. We can, for example, adapt the translation such that the queries no longer use behavioral information about the code, or we can include additional meta models and incorporate their use in the matching process. In essence, we have set up an environment where we can now easily experiment with different resolution strategies and meta models. The open-ended implementation of the translation process is an integral part of the latest version of the SOUL logic program query language [22].

4. Practical Detection of Interesting Implementation Patterns

In this section, we demonstrate the detection of some interesting implementation patterns by means of our source code templates. In particular, we apply them to the detection of accessor methods, inadvertent method invocations on null and erroneous behavior during iterations over collections. Although there exist dedicated techniques for the detection of each of these patterns, we primarily intend to demonstrate how our technique provides a uniform language to describe them all. In doing so, we illustrate how developers can use our technique to detect patterns that are specific to their development context. Each of the examples illustrates how the declarative description of a source code template can detect multiple instantiations of the same pattern, as long as the pattern adheres to the intention that is expressed in the template. The combination of behavioral and structural similarity matching is an important enabling technique in all of these examples.

4.1 Detecting and Enforcing Accessor Methods

The use of accessor methods is an example of good practice in object-oriented software development. Accessor methods provide disciplined access to the data fields contained in the objects and are

```
private Integer hour;
private Integer buffer;

public void sethour(Integer i) {
   if(i.intValue()<0 || i.intValue()>23) {
      // do something
   } else {
      hour = i;
      this.notifyDependents();
   }
}
public Integer gethour() {
   return hour;
}
public Integer gethourlazy() {
   if(hour==null)
      hour = this.currentHour();
   return hour;
}
public Integer gethourlazytoo() {
   if(hour==null) {
      hour = this.currentHour();
      return hour;
   }
   else
      return hour;
}
public Integer getBuffer() {
   Integer temp;
   temp = buffer;
   buffer = null;
   return temp;
}
public boolean setBuffer(Integer i) {
   if(buffer==null) {
      buffer = i;
      return true;
   }
   else return false;
}
```

Figure 10. Some typical accessor methods.

useful to invoke additional behavior upon each access or update to the data fields. In order for accessors to work correctly, their use should be enforced. A tool that assists in their enforcement needs to detect any references to variables that do not occur within the scope of an accessor method. However, since there are many ways to implement an accessor method, the specification of detection rules that find all these styles of accessor methods can be a hard and complex process.

In Figure 10, we show some typical accessor methods that each exhibit different structure and behavior. The `setHour` method first verifies the incoming value and, if appropriate, assigns the value to the `hour` variable. Next, it also triggers the dependency update mechanism. The `gethourlazy` method is an example of a lazy accessor method, i.e. the variable is only assigned a value upon the first execution of the accessor method. Similarly, the `gethourlazytoo` method is a variation of the same behavior, but with some different structure. Finally, the `getBuffer` and `setBuffer` methods are accessor methods for the `buffer` variable. These are yet another example of accessor methods in which we implement the behavior of a buffer: the buffer variable can only be assigned a new value if it is emptied, which occurs upon a retrieval of its value.

The detection of all these different styles of accessor methods can be relatively straightforward, as we show in Figure 11 which contains two separate queries to achieve this purpose. The first

```
1  if jtClassDeclaration(?c){
2          class ?c {
3            private ?type ?field;
4            public ?type ?name() { return ?field; }
5          }
6  }

7  if jtClassDeclaration(?c){
8          class ?c {
9            private ?type ?field;
10           public ?rt ?name(?type ?var) { ?field = ?var; }
11         }
12 }
```

Figure 11. Detect accessor methods

query (lines 1–6) detects the *getter* methods and the second query (lines 8–13) detects the *setter* methods. Both queries make use of a complete class template because they need to match both the actual accessor method as well as the declaration of the field it references. This is because the getter methods are detected as those methods that return the value of a private instance variable of the class. This can be observed on lines 3 and 4, because the *?field* logic variable must both bind to the declaration of the instance variable as well as to the expression that is returned. In other words, the field and the expression need to be the same. However, because the template is not only matched using structural matching on the source code, but also through behavioral similarity, all methods that *eventually may* return the *value* of the field are detected as getter methods. This template thus does not restrict that the method body must syntactically return the variable, but rather imposes a behavioral similarity constraint on the value returned by the method. Furthermore, the getter method's signature must have no arguments and its return type needs to be identical to the type of the field. Again, this is expressed in the template because we use the same logic variable *?type* to bind to the return type of the method and to the type of the field. This part of the template does require structural matching on the source. Likewise, the setter methods are detected as those methods that assign their sole argument to the a private field, and the type of the argument needs to be identical to the type of the field.

These queries can be further refined with additional constraints that verify the naming conventions or other possible coding conventions that are agreed upon by the development team. Furthermore, a tool can use the results of these queries to verify if all direct field accesses occur in a method that is detected as an accessor method.

4.2 Detecting Concurrent Modification Exceptions

A possible bug in Java programs happens when a modification is made to a collection that is currently being iterated over. This bug will appear upon execution of the `insertElement` method in the code of Figure 12. This method executes an iteration over a `Linkedlist` collection object during which it calls the method `operation` that adds the element to the collection. However, a collection that is under iteration may not be modified and therefore, a `ConcurrentModificationException` will be thrown and the program crashes. Obviously, in this small code snippet, the bug is easily detectable by the human developer but the same observation does not hold for large programs written by different developers. Therefore, we want to implement the automated detection of such concurrent modifications and we can do so by writing and executing a query in our tool.

The query in Figure 13 shows how we can detect one possible occurrence of the bug. It searches for all `while`-statements that use an iterator *?iterator* to loop over a collection *?collection* and

```
public List list;

public void initializeContainer() {
   List l = new LinkedList();
   list = l;
}
public void insertElement(Object x) {
   Iterator i = list.iterator();
   while(i.hasNext()) {
      Object o = i.next();
      operation(x, (Collection) this.self().list);
   }
}
public void operation(Object o, Collection c) {
    c.add(o);
}
public Example self() {
   return this;
}
```

Figure 12. Modification of a container during iteration.

```
1  if jtStatement(?s) {
2    while(?iterator.hasNext()) {
3       ?collection.add(?element);
4    }
5  },
6  jtExpression(?iterator){?collection.iterator()}
```

Figure 13. Detect additions to a container during iteration.

that perform an addition on that collection during the execution of the `while`-body. On line 3, the query also states that the value that is bound to the logic variable *?iterator* is actually an iterator object that is obtained by invoking `iterator()` on the collection object *?collection*. This particular query will thus detect the bug that is present in the code snippet of Figure 12. The template again requires a matching process that takes both structural as well as behavioral information into account. For instance, the `while`-statement can be found using mere matching on a structural meta model of the program but the call to the addition operation can occur anywhere in the control flow of the `while`-body. Furthermore, the logic variables *?collection* and *?iterator* will match with any expression that evaluates to the collection and iterator objects respectively. Once again, because of this matching process, the prototype implementation that is present in the template matches all actual places in the code where similar behavior and structure is implemented.

Of course, this template does not detect all possible occurrences of this bug. For example, we also need to detect removals of elements and take into account that there are other loop constructs available in Java. We can detect all these possibilities using multiple similar templates that each detect a possible case[5].

4.3 Finding "Inadvertent Invocation on null" bugs

A software development environment often signals the developers about incorrect syntax, inexistent method names, incorrect variable references, etc. Many development environments are even extensible in the sense that they permit to write plugins that can produce additional warnings which are otherwise only detected at compile-time or even at run-time. One such a run-time error is the invocation of a message on the `null` value. Such an error typically occurs in

[5] We are also currently working on an extension of the templates to express more variability (such as logical 'or') inside the templates themselves.

```
public void willSendToNull(Integer x) {
  if (x == null)
     this.performOperation(x);
}
private void performOperation(Integer y) {
   y.floatValue();
}
```

Figure 14. Inadvertent method invocation on null.

```
1  if jtStatement(?stat){
2       if( x == null) x.?message();
3     },
4     not(jtStatement(?stat){
5           if( x == null) { x = ?exp; x.?message();}
6         }),
7     javaMethodContainsStatement(?method,?stat)
```

Figure 15. Detect method invocations on null.

systems that use the `null` value as a special return value, indicating specific events. Figure 14 illustrates some code that features this bug. In order to prevent such errors as much as possible, we can use our approach as a plugin to a development environment and let the developers write a set of queries that try to detect such errors.

The query of this experiment, shown in Figure 15, can detect certain inadvertent invocations on the null value. It detects these by finding all method invocations on variables that are *guaranteed* to contain a null value. The first condition detects all the method invocations on a variable `x` in the `then`-branch of an `if`-statement that contains the condition `(x==null)`. Next, the second condition in the query removes all found statements where an assignment to `x` occurs on *a possible* execution path in-between the condition and the method invocation on `x`. In other words, we will not find those places in the code where the variable may be assigned a value after the `null`-check. Finally, the last line retrieves the method in which the statement occurs, which facilitates finding the statement that leads to the bug.

Once again, this example experiment shows that we are able to express the simplest implementation of the pattern we wish to detect in the source code and rely on our matching algorithm to detect all similar implementations. However, this experiment also reveals a shortcoming in the current status of our technique. It's quite clear that developers would prefer to find all places where the variable `x` *may* contain the null value. In other words, we should rather write our query such that *any possible* execution path leading to the invocation must contain an assignment to the variable. Such a semantics would detect all method invocations on variables that *may* contain a null value. Unfortunately, because of the current semantics of the matching process for statement sequences, this is not possible. We do intend to enable the implementation of such a query as part of our ongoing work with respect to the template language and the matching process.

Nevertheless, this query shows how the combination of structural and behavioral matching is necessary to easily detect certain patterns. First of all, the structural matching allows us to match the condition in an `if`-statement. Next, we need behavioral matching to detect any method invocation in the entire control-flow of the `then`-branch. Obviously, if our matching algorithm would only match the `then`-branches that directly contain a single method invocation, we would not find many bugs using this query. Because of the behavioral matching of statement sequences using the call-graph analysis, we can safely restrict the template to those statements that are essential to the pattern. Finally, although our template uses the same variable in the condition of the `if`-statement as well as in the method invocation, the actual code that will be found does not have to because we match the variables based on the points-to analysis. This means that any expression that evaluates to the same value as the variable `x` will be matched.

5. Related Work

The use of Prolog to query a base program for implementation patterns has received some attention in the past [14, 18]. ASTLOG [3] is a Prolog-based language for examining abstract syntax trees. It evaluates logic conditions against ambient parse tree nodes encountered during a tree traversal, thus avoiding the need to reify the base program as logic facts. CodeQuest [11] focuses on scalability issues by translating the Datalog subset of Prolog to SQL queries. None of these works support concrete syntax templates in logic queries, nor do they match queries against non-syntactic program representations.

Several works have been presented on the use of templates for code base querying. Works that are closely related to our approach are the LogicAJ2 [23] and Spoon [19] templates that provide a way of selecting program elements based on whether or not their implementation syntactically matches a given template. Compared to our approach, matching program elements in a syntactical way requires a template for each of the alternative ways in which a behavior can be specified. This renders the templates less expressive for finding different variations of the same pattern. Behavioral patterns are better supported in the Trace-matches [1] AspectJ extension. In it, interesting patterns on the call-graph are defined as regular expressions that are matched during the execution of the program. Trace matches are similar in spirit to our interpretation of statement sequences in source code templates. Their use of regular expressions even permits to define more complex sequences, for example allowing repetition of edges as well as optional or alternative edges. However, trace matches operate on an online dynamic analysis while our approach uses an offline call-graph analysis in combination with other representations of the program.

In the domain of program transformation, templates normally serve as a condition to a rewrite rule. In JaTS [4], a transformation is specified as a left hand side template, that must match to the elements which will be transformed, and a right hand side template which will be used instantiated to replace the matched elements. SmPL [20] follows a similar principle, it is specified not as a LHS - RHS rule, but as a Unix diff file that on a single template defines the changes that must occur on matching elements. SmPL allows for more semantic matches than JaTS by relaying on the function's control flow to match on sequences of statements, and on code isomorphisms to cope with the different ways to specify a behavior (for example, in C, `X == null` $\leftrightarrow$ `!X`). The analysis done in SmPL, however, is intraprocedural only, and does not take into account aliasing between variables on its matching; nevertheless, the use of isomorphisms permit a greater, albeit limited, degree of variability than what our approach offers. In [24], the use of concrete syntax was generalized from the Stratego program transformation language to arbitrary meta-languages. We have integrated source code templates with a logic meta language while focusing on the resolution of the resulting queries against multiple behavioral program representations.

Finally, PQL [17] is a domain specific language that uses template-like queries to match on context-sensitive traces of the program. These traces represent, for example, security flaws, violations to design rules, or possible unsafe behaviors. PQL is the closest work to ours, although it is not a complete template language in the sense that it does not represent structural code elements, thereby limiting the type of source code patterns queried to

behavioral ones. Finally, our code templates are embedded in the logic query language and rely upon full unification for the matching process, as opposed to the templates in PQL that consist of source code only.

6. Conclusion

The detection and identification of implementation patterns is an essential but often complex issue in program analysis and manipulation. At the core of this problem lies the observation that patterns do not prescribe a single implementation structure or behavior, but can rather have diverse implementation variations. Nevertheless, software developers tend to understand patterns by means of tangible examples and prototypical implementations. To achieve automatic detection, this dichotomy requires that developers convey their example-driven knowledge into operational queries or complex descriptions that effectively deal with the detection of diverse variations. This is often further complicated with the fact that the detection requires diverse structural and behavioral representations of the program. We have reconciled the example-driven knowledge of developers with the expressive power of program queries by means of embedding concrete source code templates in a logic query language. These templates permit to detect many implementation variations of patterns merely through a specification of a prototypical example implementation. We obtain this resolution strategy through an automatic translation of the template to queries that act upon diverse structural and behavioral representations.

The open-ended nature of the template translation process allows a thorough analysis of different resolution strategies and program representations in future and ongoing work. To alleviate the expressiveness problem identified in the last practical example, we intend to extend the template language with operators to indicate whether a template should match one, all or a user-defined set of possible execution paths and program contexts. To demarcate the latter, we are considering temporal logic programming which we have already successfully applied in previous work involving dynamic analysis [6]. In other previous work [5], we have experimented with fuzzy logic programming to handle in a consistent manner the impreciseness inherent to static analysis results. We intend to incorporate these approximate reasoning techniques in the template resolution strategy to compute an indication of the similarity between a template and matching base program elements.

Acknowledgments

We would like to thank the anonymous reviewers for their insightful comments.

References

[1] C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding trace matching with free variables to AspectJ. In *Proc. of the 20th Conf. on Object-oriented Programming, Systems, Languages, and Applications*, pages 345–364, 2005.

[2] J. Cohen and T. J. Hickey. Parsing and compiling using prolog. *Transactions on Programming Languages and Systems*, 9(2):125–163, 1987.

[3] R. F. Crew. ASTLOG: A language for examining abstract syntax trees. In *Proc. of the Conf. on Domain-Specific Languages*, pages 229–242, 1997.

[4] M. d'Amorim, C. Nogueira, G. Santos, A. Souza, and P. Borba. Integrating code generation and refactoring. In *Proc. of the ECOOP Workshop on Generative Programming*, 2002.

[5] C. De Roover, J. Brichau, and T. D'Hondt. Combining fuzzy logic and behavioral similarity for non-strict program validation. In *Proc. of the 8th Symp. on Principles and Practice of Declarative Programming*, pages 15–26, 2006.

[6] C. De Roover, I. Michiels, K. Gybels, K. Gybels, and T. D'Hondt. An approach to high-level behavioral program documentation allowing lightweight verification. In *Proc. of the 14th IEEE Int. Conf. on Program Comprehension*, pages 202–211, 2006.

[7] S. Ducasse, M. Freidig, and R. Wuyts. Logic and trace-based object-oriented application testing. In *Proc. of the Int. Workshop on Object-Oriented Reengineering*, 2004.

[8] J. Fabry and T. Mens. Language-independent detection of object-oriented design patterns. *Elsevier Int. Journal on Computer Languages, Systems & Structures*, 30(1-2):21–33, 2004.

[9] P. Flach. *Simply Logical*. John Wiley, 1994.

[10] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java Language Specification, Second Edition: The Java Series*. Addison-Wesley Longman Publishing Co., Inc., 2000.

[11] E. Hajiyev, M. Verbaere, and O. de Moor. Codequest: Scalable source code queries with datalog. In *Proc. of the 20th European Conf. on Object-Oriented Programming*, volume 4067 of *Lecture Notes in Computer Science*, pages 2–27, Berlin, Germany, 2006. Springer.

[12] M. Hind. Pointer analysis: haven't we solved this problem yet? In *Proc. of the 2001 Workshop on Program Analysis for Software Tools and Engineering*, pages 54–61, 2001.

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

[14] C. Krämer and L. Prechelt. Design recovery by automated search for structural design patterns in object-oriented software. In *Proc. of the 3rd Working Conf. on Reverse Engineering*, 1996.

[15] C. Lewerentz and F. Simon. A Product Metrics Tool Integrated into a Software Development Environment. In *Proc. of the ECOOP Workshop on Object-Oriented Technology*, volume 1543, pages 256–257, 1998.

[16] O. Lhoták. Spark: A flexible points-to analysis framework for java. Master's thesis, McGill University, December 2002.

[17] M. Martin, B. Livshits, and M. S. Lam. Finding application errors and security flaws using PQL: a program query language. *ACM SIGPLAN Notices*, 40(10):365–383, Oct. 2005.

[18] K. Mens, I. Michiels, and R. Wuyts. Supporting software development through declaratively codified programming patterns. In *Proc. of the 13th Int. Software Engineering and Knowledge Engineering Conf.*, 2001.

[19] C. Noguera and R. Pawlak. Open static pointcuts through source code templates. In *Proc. of the AOSD Workshop on Open and Dynamic Aspect Languages*, 2006.

[20] Y. Padioleau, R. R. Hansend, J. Lawall, and G. Muler. Semantic patches for documenting and automating collateral evolutions in linux drivers. In *Proc. of the Workshop on Linguistic Support for Modern Operating Systems*, 2006.

[21] B. G. Ryder. Dimensions of precision in reference analysis of object-oriented programming languages. In *Proc. of the 12th Int. Conf. on Compiler Construction*, volume 2622, pages 126–137, 2003.

[22] The Smalltalk Open Unification Language (SOUL). http://prog.vub.ac.be/SOUL/.

[23] M. A. Tobias Rho, Günter Kniesel. Fine-grained generic aspects. In *Proc. of the AOSD Workshop on Foundations of Aspect-Oriented Languages*. 2006.

[24] E. Visser. Meta-programming with concrete object syntax. In *Generative Programming and Component Engineering*, volume 2487 of *Lecture Notes in Computer Science*, pages 299–315. Springer-Verlag, 2002.

[25] R. Wuyts. *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation*. PhD thesis, Vrije Universiteit Brussel, Belgium, January 2001.