A Logic Meta-Programming Foundation for Example-Driven Pattern Detection in Object-Oriented Programs

Coen De Roover
Software Languages Lab
Vrije Universiteit Brussel, Belgium
cderoove@vub.ac.be

***Abstract*—This paper summarizes the doctoral dissertation in which we introduced an example-driven approach to pattern detection. This approach enables specifying pattern characteristics in a familiar language: through a code excerpt that corresponds to their prototypical implementation. Such excerpts are matched against the program under investigation according to various matching strategies that vary in leniency. Each match is quantified by the extent to which it exhibits the exemplified characteristics. The smaller this extent, the more likely the match is a false positive —thus establishing a ranking which facilitates assessing a large amount of matches. Unique to the matching process is that it incorporates whole-program analyses in its comparison of individual program elements. This way, we are able to recall implicit implementation variants (i.e., those implied by the semantics of the programming language) of a pattern of which only the prototypical implementation has been exemplified.**

I. Introduction

Testament to their valuable applications within quality assurance, is the growing amount of tools (e.g., *CodeQuest* [1], *JQuery* [2], *Metal* [3], *Condate* [4], *JTL* [5], *PQL* [6], *.QL* [7]) that identify source code exhibiting characteristics of interest. We refer to such tools as pattern detection tools.

A pattern's characteristics can be structural as well as behavioral. While the former concern instructions and their organization (e.g., inter-class relationships such as inheritance), the latter concern the order in which instructions are executed and the values operated on by these instructions (i.e., control flow and data flow). Reading from a closed file, for instance, is characterized behaviorally by two instructions `x.close()` and `y.read()` that are executed sequentially and operate on the same file. Possible violations of the invariant that equal objects must have equal hash codes, in contrast, are characterized structurally by a custom `hashCode()` method without a corresponding `equals(Object)` method.

Tools that support detecting user-specified patterns provide application-specific support for the development process. They enable a development team to enforce its coding conventions, to detect violations against the protocol of an in-house API and even to check whether there are similar instances of a bug upon its discovery. Despite their valuable applications, such pattern detection tools have yet to become an integral part of every developer's toolbox. Through an extensive survey, our dissertation [8] identified several shortcomings of the state-of-the-art this can be attributed to.

First of all, a pattern's characteristics have to be specified in highly specialized languages that are unfamiliar to most developers. For control flow characteristics alone, specification languages already include reachability queries [9], [4], plain [10] as well as temporal logic formulas [11] and state machines [3] that quantify over control flow graphs. Moreover, most languages support only the characteristics that are essential to their intended use. Other characteristics are not supported at all —even though many patterns are heterogeneously characterized. There is therefore a need for a *familiar* specification language in which structural and behavioral characteristics can be specified *uniformly*.

Secondly, we identified the need for a detection mechanism that recalls implicit *implementation variants* of specified characteristics. This relieves users from having to enumerate each variant in a specification. Recent advances in program analysis have enabled detecting these variants in industrially-sized programs. However, different analyses implement different trade-offs with respect to precision and analysis time. For instance, many alias analyses are only able to assert conservatively that two expressions may alias during a possible program execution —in reality, these expressions might not alias at all. As a result, detailed knowledge about a tool's enabling analyses is required to assess the extent to which a reported pattern instance exhibits the specified characteristics. There is therefore also a need to *facilitate user assessment* of the reported pattern instances.

In the dissertation [8], we presented an example-driven approach to pattern detection that addresses these shortcomings of the state of the art. Its specification language enables exemplifying a pattern through familiar code excerpts that correspond to its prototypical implementation. Its detection mechanism recalls implicit implementation variants of the exemplified pattern and ranks these according to the extent that they exhibit the pattern's characteristics.

II. Overview of the Approach

Our approach is based on four cornerstones: *logic meta-programming*, *example-based specification*, *domain-specific unification* and *fuzzy logic*. We refer to the dissertation for

a discussion of the fifth and crosscutting *open implementations* cornerstone. Each cornerstone provides a meta-interface through which it can be extended at a higher level of abstraction than its implementation.

A. Logic Meta-Programming

Logic meta-programming [12] (LMP) is the founding cornerstone of our approach. It advocates using formulas in an executable logic as specifications of a pattern's characteristics. This merely requires reifying the program under investigation such that variables can range over its elements. Executing a proof procedure will establish whether program elements exhibit the characteristics specified in a formula. LMP is widespread in the literature (e.g., [13], [12], [2], [14], [10] and [6], [1], [5] use variants of Prolog and Datalog respectively) for structural or behavioral pattern characteristics. This tendency somewhat lessens the burden that the diversity among specification languages poses to users.

The declarative nature of logic programming should result in expressive and descriptive pattern specifications. In practice, however, such specifications tend to become convoluted and operational in nature. To illustrate, consider a pattern that describes a coding convention in which `Component` subclasses (except the one named `Composite`) are required to define a method `acceptVisitor(ComponentVisitor)` that logs a message before double dispatching to its parameter. The top-right corner of Figure 1 depicts the prototypical implementation of a complying class.

The top-left corner of the same figure depicts an LMP specification for this pattern. As indicated by the `if`-keyword, it is a logic query (i.e., a logic formula for which variable bindings have to be found such that the formula holds). The syntax[1] stems from SOUL [15], the concrete LMP instance on which we have founded our prototype. This query quantifies explicitly over the AST nodes of the program under investigation. In each solution to the query, logic variable `?class` is bound to a class declaration AST node that complies with the coding convention.

Note that the conditions on lines 4–13 merely have to be negated to find class declaration AST nodes that violate the coding convention. However:

- The query is *convoluted*. Lines 1–4 identify a method `?m` defined in a class `?class` that extends the fully qualified type `example.Component` and is not named `Composite`. Lines 5–8 ensure that method `?m` is named `acceptVisitor` and has a single parameter that can be referred to by the name `?id` in its body `?body`. Lines 9–13 ensure that this body consists of two statements wrapping the actual expressions `?log` and `dd` that perform the logging and the double dispatching respectively. Line 13 specifies how the latter has to be implemented: by means of an invocation of which the receiver is named `?id` (i.e., the parameter of the method) and of which the single argument is an unqualified `this`-expression (i.e., `this`).
- *Expert knowledge* is required to understand this query. First of all, one must know how the program representation is reified in order to quantify over it. In this case, AST nodes seem to be reified as compound terms [2] (e.g., `simpleName(?id)`) of which each argument is a reified child node. Second, one must know how the relations between reified program elements are reified (e.g., the two-argument predicate `methodDeclarationHasName:/2`). Finally, one has to be aware of the intricate details of the program representation itself. In this case, the details of the abstract grammar for the actual AST nodes (e.g., expressions within a block are wrapped in a statement) and the parser that produced them (e.g., `System.out` can be parsed as a field access or as qualified reference). Both stem from the Eclipse JDT.
- The query *recalls only the pattern's prototypical implementation* and not a single of its implementation variants depicted on the right-hand side of Figure 1. For instance, line 13 requires the argument to the second statement in the method to be a `this`-expression rather than an arbitrary expression that evaluates to the current object. The query therefore fails to recognize `MustAliasLeaf` as a complying class, even though the required double dispatching is implemented through a temporary variable.

To recall implementation variants, an LMP query either has to enumerate them or specify their shared machine-verifiable behavior. The former queries quantify over reified AST nodes, but only recall the enumerated variants. The latter queries recall all implementations of the specified behavior, but do so by quantifying over the reified results of program analyses. This exposes users to their details —which are more intricate than those of AST nodes illustrated above. Alternatively, analysis results can be quantified over using a dedicated language (e.g., temporal logic over CFGs). However, most are in stark contrast with the way developers tend to communicate about patterns; in terms of class diagrams and code snippets that exemplify their prototypical implementation.

As the founding cornerstone of our approach, LMP lends its means for expressing explicit points of variation among pattern instances (i.e., logic variables and connectives). It also lends its provisions for abstraction and reuse (i.e., predicate definition). The remaining cornerstones remedy the aforementioned shortcomings of LMP.

[1] In SOUL, the syntax for a predicate closely resembles the one of Smalltalk for a message sent to the first argument of the predicate. SOUL retains the traditional notation for compound terms (i.e., a functor symbol followed by its arguments). Logic variables are preceded by a question mark. Logic lists are demarcated by angle brackets. Examples include the empty list `<>`, the list with three elements `<1,2,3>` and every list with head `?h` and tail `?t`: `<?h|?t>`. The SOUL condition on line 6 in the top-left corner of Figure 1 would therefore be equivalent to the Prolog condition `methodDeclarationHasParameters(M,nodeList([P]))`.

[2] SOUL relies on a linguistic symbiosis with Java to reify AST nodes. The reified version of an AST node is the AST node itself (i.e., an instance of `org.eclipse.jdt.core.dom.ASTNode`). This facilitates exploiting query results in other tools. The domain-specific unification procedure (cf. Section II-C) ensures that reified AST nodes unify with structurally equivalent compound terms.

Fig. 1. Example-driven pattern detection motivated through the shortcomings of the plain logic meta-programming approach to pattern detection.

B. Example-Based Specification

The example-based specification cornerstone enables exemplifying the prototypical implementation of a pattern's characteristics. Within logic formulas, this implementation can be exemplified as a code excerpt in the concrete syntax of the program under investigation. This obviates the need to quantify explicitly over the reified program representation to express such characteristics. Developers are therefore shielded from the details of the program representation and its reification.

The bottom-left corner of Figure 1 depicts the example-based specification for a class that complies with the aforementioned coding convention. In contrast to the LMP specification, this specification is concise and descriptive. It closely resembles the prototypical implementation of a complying class. Apart from logic variables, only a negation operator (i.e., `!Composite` to exclude classes named `Composite`) and a reflexive transitive closure operator (i.e., `extends*` to include classes that extend `Component` indirectly) has been added.

We instantiated the example-based specification cornerstone as so-called *template terms* [16] in SOUL. The template term in Figure 1 consists of a functor `jtClassDeclaration`, a single argument `?class` and a code excerpt demarcated by braces. The functor identifies the grammar rule adhered to by the code excerpt. This grammar describes the concrete syntax of Java —extended with a minimum of non-native syntax and logic variables. The latter stand for productions that originate from a non-terminal in the Java grammar. Used as a condition, a template term succeeds if there is an AST node that matches the code excerpt. Backtracking over the term successively unifies each matching node with the argument of the term. Variables within the excerpt get bound as well.

Support for concrete syntax is not uncommon among advanced pattern detection tools (e.g., [3], [14], [4], [6], [17]). In contrast to these approaches, template terms adopt source code excerpts of a coarse granularity. The concrete syntax of whole method and class declarations can be used. For matching such template terms, we do not limit ourselves to the strictly syntactic strategy that is predominant. Instead, we match according to multiple strategies that vary in leniency. All of them are considered when the template term is backtracked over. As they realize the example-based semantics of template terms, we refer to these matching strategies as example-based interpretations. The following example-based interpretations are predefined:

- Under the *syntactic interpretation*, AST nodes match a template term if they exhibit the structural characteristics exemplified by its code excerpt. Moreover, matching nodes should not exhibit any other structural characteristics. Only class `PrototypicalLeaf` from Figure 1 matches the template under this interpretation.
- The *lexical interpretation* is a less restrictive version of the syntactic one. Matches have to exhibit the structural characteristics exemplified by the template, but are allowed to exhibit additional ones.[3] Class declarations, for instance, are allowed to have additional declarations besides the exemplified ones.
- Under the *control flow interpretation*, matching method declarations have to exhibit the control flow characteristics exemplified by the template term. There should be a path through the control flow graph of the method (i.e., existentially qualified) on which all exemplified instructions are executed. Non-specified instructions are allowed on the execution path. The path also crosses method boundaries (i.e., it is inter-procedural). As a result, matches for an instruction in the template term need not reside in the method declaration that matches the term.[4] This is why class `SuperLogLeaf` from Figure 1 matches the template term under this interpretation.

[3]However, the lexical relations among the elements of a match have to be the same as the ones among the corresponding elements in the template. If a statement in the excerpt is preceded by a local variable declaration, for instance, matching statements have to be preceded by a matching variable declaration as well.

[4]However, actual statements such as `return`-statements are matched intra-procedurally.

In template terms, multiple occurrences of the same variable express a data flow characteristic. For instance, the occurrences of `?v` in Figure 1 link the receiver of `?v.visitMethod(this)` to the parameter of method `acceptVisitor(?t ?v)`. Their bindings have to unify —which brings us to the third cornerstone of our approach.

C. Domain-Specific Unification

Unification is an essential ingredient of the proof procedure in logic programming. The domain-specific unification procedure [18] treats reified program elements different from other terms. Unifying two reified program elements can succeed where the general-purpose unification procedure fails. The table in Figure 2 lists the most important extensions.

The first extension ensures that reified AST nodes unify with structurally equivalent compound terms, even if they have not been reified as such. [2] The remaining extensions are specific to the pattern detection domain. Implicit implementation variants (i.e., those implied by the semantics of the programming language) of the same pattern characteristic unify. This obviates the need to enumerate these variants in a specification. To this end, the domain-specific procedure consults *whole-program* analyses when unifying *individual* reified program elements.

For instance, a semantic analysis determines whether an unqualified and fully qualified type should unify. An alias analysis determines whether two expressions should unify. This allows syntactic deviations as long as the expressions may evaluate to the same object at run-time. Method invocation and declaration names unify if the invocation may invoke the declaration. Users benefit from the results of these analyses without being exposed to their intricate details.

D. Fuzzy Logic

A fuzzy variant of SOUL [19], close to f-Prolog [20], lends our detection mechanism a means to rank the results it reports. Each result is quantified by the extent to which it exhibits the characteristics in a specification. The smaller this extent, the more likely the reported instance is a false positive.

Concretely, truth degrees are associated with each match for a template term. These are bounded by the example-based interpretation under which the match was found. They cannot exceed 1, $\frac{9}{10}$ and $\frac{8}{10}$ for the syntactic, lexical and control flow interpretation respectively. The properties of the match itself further refine this upper bound. Matches are ranked lower if they required a unification that could introduce false positives. To this end, unification degrees are associated with each domain-specific extension (cf. Figure 2). They reflect the likelihood that such an extension may introduce false positives due to imprecisions in its enabling analysis.

Most notably, expressions unify with a unification degree of $\frac{1}{2}$ if they may alias according to an inter-procedural points-to analysis. Unifying such expressions can introduce false positives if the expressions do not evaluate to the same object during all possible program executions. Expressions that reside in the same method unify with a unification degree of $\frac{9}{10}$ if they are guaranteed to alias during all executions according to an intra-procedural must-alias analysis. If both expressions are the same AST node, they unify with a degree of 1.

The screenshot in Figure 1 illustrates that our detection mechanism recalls all implementation variants of the coding convention. Class `PrototypicalLeaf` is ranked highest as it exhibits the exemplified characteristics to the greatest extent. Next is class `SuperLogLeaf` which could only be detected by following its super method invocation, but without having to consult a program analysis. Class `MayAliasLeaf` is ranked lowest as it only preforms the required double dispatching when the user inputs an even number. Its detection required consulting program analyses.

III. AN ILLUSTRATIVE EXAMPLE

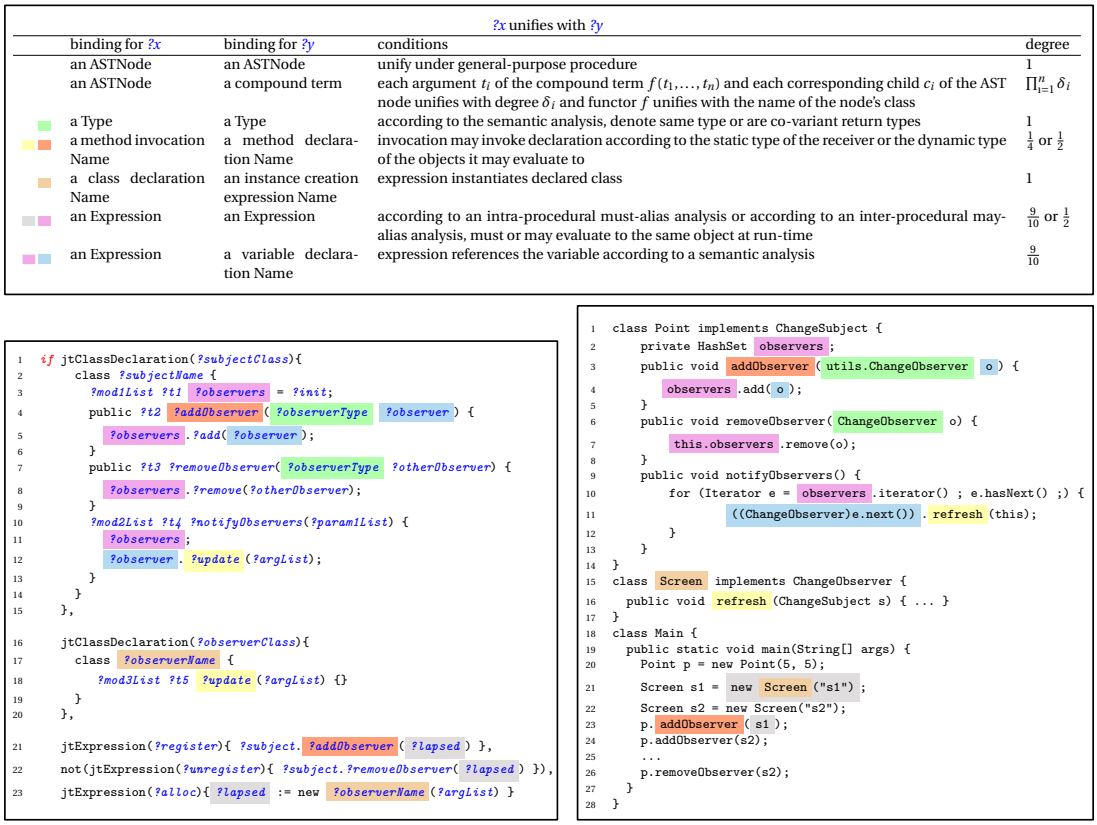
We briefly illustrate our approach by applying its instantiation in SOUL to instances of the *Observer* design pattern and the *lapsed listener* pitfall in their implementation.

Detecting Observers Lines 1–20 in the bottom-left corner of Figure 2 depict an example-based specification for the *Observer* pattern. The depicted specification consists almost entirely of logic variables. These indicate explicit points of variation that are constrained by the unification procedure. We have highlighted the occurrences of the same variable in a distinct color. When substituting for an expression or a variable declaration, these express data flow characteristics.

Lines 1–15 exemplify the subject participant as a class with a collection of `?observers` to which an `?observer` can be added through method `?addObserver`. Note that we have used different variables for the parameters of `?removeObserver` and `?addObserver`. Otherwise, the specification would require that at least one observer is added to and removed from a subject at run-time. This is because our unification procedure unifies parameters only if they are in a may-alias relation. Method `?notifyObservers` notifies the subject's observers of a state change. Rather than enumerating the different ways in which the `?observers` field can be iterated through, the specification exemplifies the method as one with two successively evaluated instructions. The first evaluates to the `?observers` field, while the second sends message `?update` to an `?observer` that has been added to this field through method `?addObserver`. Lines 16–20 exemplify the observer participant as a class in which the invoked `?update` method resides. Note that this already constrains `?observerClass` to a class declaration in the subtype hierarchy of `?observerType`. No additional conditions are therefore required to express this constraint.

A matching *Observer* implementation is depicted next to this specification. To highlight the domain-specific unifications its detection required, the colors in both figures correspond. For instance, the occurrences of `?observer` substitute for expressions as diverse as `((ChangeObserver)e.next())` and the parameter of method `addObserver` —both deemed in a may-alias relation though.

Detecting Lapsed Listeners *Lapsed listeners* are observer participants that are no longer needed, but never unregister from their subject. Lines 1–20 of the specification exemplify the classes that participate in the design pattern (i.e.,



?x unifies with ?y

binding for ?x	binding for ?y	conditions	degree
an ASTNode	an ASTNode	unify under general-purpose procedure	1
an ASTNode	a compound term	each argument t_i of the compound term $f(t_1,\ldots,t_n)$ and each corresponding child c_i of the AST node unifies with degree δ_i and functor f unifies with the name of the node's class	$\prod_{i=1}^{n}\delta_i$
a Type	a Type	according to the semantic analysis, denote same type or are co-variant return types	1
a method invocation Name	a method declaration Name	invocation may invoke declaration according to the static type of the receiver or the dynamic type of the objects it may evaluate to	$\frac{1}{4}$ or $\frac{1}{2}$
a class declaration Name	an instance creation expression Name	expression instantiates declared class	1
an Expression	an Expression	according to an intra-procedural must-alias analysis or according to an inter-procedural may-alias analysis, must or may evaluate to the same object at run-time	$\frac{9}{10}$ or $\frac{1}{2}$
an Expression	a variable declaration Name	expression references the variable according to a semantic analysis	$\frac{9}{10}$

```
1  if jtClassDeclaration(?subjectClass){
2      class ?subjectName {
3        ?mod1List ?t1 ?observers = ?init;
4        public ?t2 ?addObserver(?observerType ?observer) {
5          ?observers.?add(?observer);
6        }
7        public ?t3 ?removeObserver(?observerType ?otherObserver) {
8          ?observers.?remove(?otherObserver);
9        }
10       ?mod2List ?t4 ?notifyObservers(?param1List) {
11         ?observers;
12         ?observer.?update(?argList);
13       }
14     }
15   },

16   jtClassDeclaration(?observerClass){
17     class ?observerName {
18       ?mod3List ?t5 ?update(?argList) {}
19     }
20   },

21   jtExpression(?register){ ?subject.?addObserver(?lapsed) },
22   not(jtExpression(?unregister){ ?subject.?removeObserver(?lapsed) }),
23   jtExpression(?alloc){ ?lapsed := new ?observerName(?argList) }
```

```
1  class Point implements ChangeSubject {
2      private HashSet observers;
3      public void addObserver(utils.ChangeObserver o) {
4          observers.add(o);
5      }
6      public void removeObserver(ChangeObserver o) {
7          this.observers.remove(o);
8      }
9      public void notifyObservers() {
10         for (Iterator e = observers.iterator() ; e.hasNext() ;) {
11             ((ChangeObserver)e.next()).refresh(this);
12         }
13     }
14 }
15 class Screen implements ChangeObserver {
16   public void refresh(ChangeSubject s) { ... }
17 }
18 class Main {
19   public static void main(String[] args) {
20     Point p = new Point(5, 5);
21     Screen s1 = new Screen("s1");
22     Screen s2 = new Screen("s2");
23     p.addObserver(s1);
24     p.addObserver(s2);
25     ...
26     p.removeObserver(s2);
27   }
28 }
```

Fig. 2. Domain-specific extensions of the unification procedure illustrated on the detection of lapsed listeners in the *Observer* design pattern.

`?subjectClass` and `?observerClass`). Lines 21–23 exemplify the lapsed listener pitfall at the instance-level: as instances of the participating classes that exhibit the characteristics of the pitfall. They identify `?lapsed` objects that are added to a `?subject` (line 21), but never removed from it (line 22). The final condition term is optional. It identifies the expression that instantiated the lapsed object. To this end, it uses the non-native operator `:=` which unifies the logic variable on its left-hand side with the AST node that matches the code on its right-hand side. As a result, `?alloc` will be bound to `new Screen("s1")` for the depicted program.

Note that the depicted specification only detects possible lapsed listeners. It does not identify the point in the program's execution after which an observer is no longer needed, nor does it specify that the `?unregister` expression should be executed after the `?register` expression. It can therefore only be used to issue warnings. Actual lapsed listeners could be identified through a subsequent dynamic analysis.

IV. Evaluation

In the dissertation [8], we formulated several well-motivated desiderata for each dimension in the design of a pattern detection tool: its specification language, its detection mechanism and its program representation. When fulfilled, these result in a *general-purpose* tool that can be applied to detect *structural and behavioral* pattern characteristics using *descriptive specifications* in a *uniform* language. Through running examples, we motivated each individual cornerstone of our approach using the desiderata it helps to fulfill. Next, we evaluated our approach as a whole on these desiderata by detecting patterns that are representative for the intended use of a general-purpose tool: design patterns, μ-patterns and bug patterns.

Overall, the patterns were straightforward to specify in an example-driven manner. Even though the patterns are diverse and heterogeneously characterized, their specifications are descriptive. The majority consists of exclusively template terms with little non-native syntax. We had to resort to higher-order logic predicates only for cardinality constraints such as "*for-all*" and "*as-many-as*". We were able to detect most of the pattern instances with few false positives. Recalling the missing instances would require exemplifying additional prototypical implementations of the pattern —of which the detection mechanism recognizes implicit implementation variants. Many false positives could be eliminated by adding logic conditions to the specification that implement heuristics.

V. Future Work

There are still many open questions related to our approach. For instance, the ranking of the detection results was intended to reflect their projected likelihood of being a false positive. Large corpus-based studies are needed to test these projections against reality. Using UML-like diagrams instead of code templates to exemplify a pattern would be a natural extension of our approach. Interestingly, logic variables could serve as links between a class diagram and a sequence diagram in such specifications. We have already started to explore this

path. The most challenging question concerns methods for generalizing concrete instances into an example-based pattern specification that is no more and no less permissive than intended. Perhaps the corresponding search space could even be explored automatically. To our knowledge, the resulting tool would be unique given that example-based specifications are backed by advanced program analyses.

Extensive corpuses of software idioms, bug patterns and design patterns are still lacking. Corpuses of design pattern instances exist, but their coarse-grained nature renders assessing detection results difficult. For example, they do not contain information about which operations are called in a *Template Method*.

From a broader perspective, it will become increasingly important to investigate how the valuable software engineering tools that we develop as a community can become an integral part of every developer's toolbox. In a sense, our own work attempts to vulgarize program analyses in the hope that developers start applying them to everyday software engineering problems.

VI. Lessons Learned

I conclude with some personal reflections for the PhD symposium. What sped up my PhD process significantly was a conscious decision not to reinvent the wheel, but to stand on the shoulder of giants instead. I did not implement a logic programming language from scratch, but rather incorporated my ideas into SOUL. Besides saving time, this also exposed the ideas to the other researchers working on or with SOUL. A joint research artefact easily gathers momentum. Likewise, I decided to rely on the Eclipse JDT for structural program information and on the SOOT analysis framework for behavioral program information. This was not an obvious choice though. Live instances of these Java programs had to be interacted with, but SOUL is implemented in Smalltalk. Some perseverance (and the JAVACONNECT Smalltalk-to-Java interconnection library) was necessary to make this work. To make a long story short, I'm greatly indebted to all researchers that made their artifacts publicly available. On the other hand, sometimes it takes implementing an algorithm yourself to fully grasp all of its details.

That said, it is important not to spend time on the wrong details. I devoted a substantial amount of time devising example-driven specifications for every single μ-pattern. Validation-wise, however, this contributed little to the dissertation. In the last year of my research, analysis paralysis was always looming. To others susceptible to this terrible infliction I can only advise to trust your advisors when they say you have enough material and write everything down as quickly as possible. To put everything into perspective, an anonymous researcher once told me that a PhD is akin to a driver's license. In the end, it only marks the beginning of your academic career.

Acknowledgment

I would like to thank the promotor and co-promotor of my dissertation for their support: Wolfgang De Meuter and Johan Brichau. Coen De Roover is funded by the *Stadium* SBO project sponsored by the "Flemish agency for Innovation by Science and Technology" (IWT Vlaanderen).

References

[1] E. Hajiyev, M. Verbaere, and O. de Moor, "CodeQuest: Scalable source code queries with Datalog," in *Proceedings of the 20th European Conference on Object-Oriented Programming (ECOOP06)*, ser. Lecture Notes in Computer Science, vol. 4067, 2006, pp. 2–27.

[2] K. De Volder, "JQuery: A generic code browser with a declarative configuration language." in *Proceedings of the 8th International Symposium on Practical Aspects of Declarative Languages (PADL06)*, 2006, pp. 88–102.

[3] D. Engler, B. Chelf, A. Chou, and S. Hallem, "Checking system rules using system-specific, programmer-written compiler extensions," in *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI00)*, Oct. 2000.

[4] N. Volanschi, "A portable compiler-integrated approach to permanent checking," in *Proceedings of the 21st IEEE International Conference on Automated Software Engineering (ASE06)*, 2006, pp. 103–112.

[5] T. Cohen, J. Y. Gil, and I. Maman, "JTL: the Java Tools Language," in *Proceedings of the 21st ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA06)*, 2006, pp. 89–108.

[6] M. Martin, B. Livshits, and M. Lam, "Finding application errors and security flaws using PQL: a program query language," in *Proceedings of the 20th ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages and Applications (OOPSLA05)*, 2005, pp. 365–383.

[7] O. de Moor, M. Verbaere, E. Hajiyev, P. Avgustinov, T. Ekman, N. Ongkingco, D. Sereni, and J. Tibble, "Keynote address: .ql for source code analysis," in *Proceedings of the 7th. IEEE International Working Conf. on Source Code Analysis and Manipulation (SCAM07)*, 2007, pp. 3–16.

[8] C. De Roover, "A logic meta programming foundation for example-driven pattern detection in object-oriented programs," Ph.D. dissertation, Vrije Universiteit Brussel, August 2009.

[9] S. Drape, O. de Moor, and G. Sittampalam, "Transforming the .NET intermediate language using path logic programming," in *Proceedings of the 4th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'02)*, 2002, pp. 133–144.

[10] H. Falconer, P. H. J. Kelly, D. M. Ingram, M. R. Mellor, T. Field, and O. Beckmann, "A declarative framework for analysis and optimization," in *Proceedings of the 16th International Conference on Compiler Construction (CC07)*, 2007.

[11] D. Lacey, "Program transformation using temporal logic specifications," Ph.D. dissertation, University of Oxford, August 2003.

[12] R. Wuyts, "A logic meta-programming approach to support the co-evolution of object-oriented design and implementation," Ph.D. dissertation, Vrije Universiteit Brussel, Belgium, January 2001.

[13] R. F. Crew, "ASTLOG: A language for examining abstract syntax trees," in *Proceedings of the 1997 USENIX Conference on Domain-Specific Languages (DSL'97)*, 1997, pp. 229–242.

[14] M. Appeltauer and G. Kniesel, "Towards concrete syntax patterns for logic-based transformation rules," in *Proceedings of the Eighth International Workshop on Rule-Based Programming (RULE07)*, 2007.

[15] "The SOUL Website," http://soft.vub.ac.be/SOUL/, 2011.

[16] C. De Roover, J. Brichau, C. Noguera, T. D'Hondt, and L. Duchien, "Behavioral similarity matching using concrete source code templates in logic queries," in *Proceedings of the Symp. on Partial Evaluation and semantics-based Program Manipulation (PEPM07)*, 2007, pp. 92–102.

[17] E. Visser, "Meta-programming with concrete object syntax," in *Proceedings of the 1st Conf. on Generative Programming and Component Engineering (GPCE02)*, 2002, pp. 299–315.

[18] J. Brichau, C. De Roover, and K. Mens, "Open unification for program query languages," in *Proceedings of the XXVI International Conference of the Chilean Computer Science Society (SCCC 2007)*, 2007.

[19] C. De Roover, J. Brichau, and T. D'Hondt, "Combining fuzzy logic and behavioral similarity for non-strict program validation," in *Proceedings of the 8th ACM-SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP06)*, 2006, pp. 15–26.

[20] D. Li and D. Liu, *A fuzzy Prolog database system*. John Wiley & Sons, Inc., 1990.